MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN SOFTWARE ENGINEERING

Lightweight Data Sharing System based on Bidirectional Transformations

Duchêne, Adrien; Marchal, Hugues

Award date:
2018

Awarding institution:
University of Namur

Université de Namur
Faculty of Computer Science
Academic Year 2017–2018

Lightweight Data Sharing System based on Bidirectional Transformations

Adrien DUCHÊNE Hugues MARCHAL

Internship mentor: Zhenjiang Hu

Supervisor: _____ (Signed for Release Approval - Study Rules art. 40)
Pierre-Yves Schobbens

A thesis submitted in the partial fulfillment of the requirements
for the degree of Master of Computer Science at the Université of Namur

Abstract

Although the data sharing and synchronizing problems have been raised many years ago, they remain major issues in the database community. Still, some tools are provided to end-users in order to answer some of their needs. Yet, those platforms are most likely very complicated to handle notably because they ask the user to have very much knowledge, the user sometimes being the developer. Also, most of those systems do not really ensure data consistency. Our approach based on bidirectional transformations (BXs) resolves collaboration between companies having their own data structure in an easier way, improving the data consistency guarantees. This means that the user does not need to know the structure of the data other than his and the shared mappings, and will also never be asked to use pure code or database knowledge, limiting then the complexity. In addition to this, the system uses the bidirectional transformations properties to authorize or not editing the shared data as BiGUL has indeed such power for any user in the sharing group. Moreover, the system is extensible in the way that the user can easily join a sharing group, after providing to the bidirectional functions a mapping table, similar to a GLAV (Global-Local-As-View) mapping matching his local structure with the shared one.

Bien que les problèmes de partage et synchronisation de données aient été soulevés il y a un certain temps, ils restent encore à l'heure actuelle un problème majeur au sein de la communauté des bases de données. Il existe néanmoins différents outils répondant à certains besoins des utilisateurs finaux. Malgré tout, ces plateformes sont généralement très compliquées à prendre en main, principalement puisqu'elles requièrent beaucoup de connaissances de la part de l'utilisateur, qui est parfois le développeur. Aussi, la majorité de ces systèmes ne garantissent pas vraiment la consistance des données. Notre approche, basée sur les transformations bidirectionnelles (BXs), apporte une solution pour une collaboration plus facile entre des entreprises possédant leur propre structure de données tout en augmentant les garanties de consistance de données. L'utilisateur ne doit donc pas connaître de structure autre que la sienne et que le format partagé. Il ne lui sera pas non plus demandé d'utiliser du code pur ni des connaissances poussées en bases de données, limitant ainsi la complexité. En outre, le système tire parti des propriétés des transformations bidirectionnelles codées en BiGUL pour autoriser ou non l'édition des données par tout membre du groupe de partage. De plus, le système est extensible dans le sens où un utilisateur peut aisément rejoindre un groupe de partage, après avoir donné aux fonctions bidirectionnelles la table de mapping, laquelle est similaire à un mapping GLAV (Global-Local-As-View) appariant sa structure locale à la structure partagée.

Acknowledgements

We would like to thank our supervisor, Pierre-Yves Schobbens, for the various tips and comments he gave us, mainly for the different rereadings of this thesis. We would also like to warmly thank professor Hu for the great feedback and motivation that he gave us during the whole internship and beyond. It helped us to complete this work notably leading to the publication of a paper, which he revised, for the "Programming 2018" BX conference held in Nice. Hsiang-Shang "Josh" Ko is also to be thanked for his contributions in our work, thanks to his deep understanding of bidirectional transformations and BiGUL. All the members of the PRL lab, and mainly Zirun Zhu must also be thanked for his advices on Haskell.
Finally, we would like to thank the NII and the UNamur for the great opportunity they offered us.

Contents

Glossary

DBMS : A database management system (DBMS) is a software system for creating and managing databases.

PDBMS : A PDBMS consists of three main components: an interface, a peer-to-peer layer, and a DBMS.

API : A set of procedures/functions allowing the creation of applications that access the features or data of an operating system.

SPOF : A single point of failure (SPOF) is a part of a system that, if it fails, will stop the entire system from working.

TTL : A time to live (TTL) indicates the time during information must be kept.

NAT : A network address translator (NAT) is a method to map internet IP addresses with Intranet IP addresses.

Introduction

Two companies working for example in the aviation field that have a collaboration agreement may want to share their data and/or results. Yet, the two companies might be different in the way they operate, for storing data notably. Because of different but still related stored data and of differences in the information treatment, the structure of the database is unlikely to be the same. Despite all those differences, they still want to be able to share relevant and consistent data without having to know every technical aspect of the sharing process, including the different structures. The two companies then want have a direct access to the data of the other company. Yet, to avoid many database connections and data transfers such as with Foreign Data Wrappers (FDW) [23], the two companies may prefer to merge the foreign data directly inside their database, as some SQL views synchronizations do.

This problem of sharing data among companies has been raised many years ago. In 1997 is proposed "*a data processing method for synchronizing the data records of a plurality of disparate databases*" [25]. A year later is proposed a concrete system and a method to synchronize the data of a central computer having a central database with one or more other computers [31].
It then proves that this problem has already been answered in some ways. Nowadays, different approaches to manage sharing or/and synchronizing[1] data already exist, such as SQL Views synchronization methods [33, 8, 9, 10, 5, 1, 7, 6], Foreign Data Wrappers (FDW) [23], the Hyperion Project [12], ORCHESTRA [21] and PeerCSDB [41], for example. Yet, most platforms are not very simple, either by requiring too much information from the user, or by asking him to know how to code or how are all the databases structured. Also, most of those platforms cannot really guarantee the consistency of the data they share or/and synchronize. Still, this last concern has already been taken into account, notably by the "Relational Lenses" [14] platform and "BRUL" [46], a library providing the user an easier way to code the putback function with flexible update policies. The two approaches, among many others [45, 36, 47], use bidirectional transformations. Nevertheless, they remain quite difficult to handle, as mentioned before.

Let us continue, with more details, the example started above. There were two companies working in the aviation field, namely AA and BA. For the purpose of this example, let us say that AA bought BA, which explains why BA's database may contain different data and how they are related. In this context, AA wants to complete its list of flights including the ones of company BA. Yet, as it is possible to see in Figure 0.1, the structure of the tables are different. A mapping, shown in Figure 0.2, is necessary to match the data of company AA to the data of company BA. As the goal of AA is to store all the flights that the company proposes, the data merging process needs to check that there are no duplicated entries in the table. If, thanks to the mapping table, an entry is matched to

[1] Note that in general, synchronization and sharing are different concepts. Bidirectional transformations being able to deal with authorizations, in this context, sharing also includes synchronization.

AA_Passenger

pid	name
1	Renee
2	Verena
3	Iluju

AA_Ticket

pid	fno	meal
1	AA210	Meat
2	AA378	Veg.

AA_Flight

fno	date	dest	sold	cap
AA210	01/05/03	L.A.	120	256
AA341	01/15/03	N.Y.	160	160
AA378	01/21/03	S.F.	90	124

(a) AA's data

BA_Passenger

pid	name
1	John
2	Renee

BA_Fleet

aid	type	capacity
B-1	Boeing 747	340
B-2	Boeing 737	130
B-3	Boeing 737	107

BA_Flight

fno	date	to	sold	aid
BA1023	01/07/03	LAX	67	B-3
BA1078	01/15/03	JFK	118	B-2
BA1109	01/15/03	ORD	164	B-1

BA_Reserve

pid	fno
1	BA1023
2	BA1078
2	BA1109

(b) BA's database

Figure 0.1: Data present in AA (0.1a) and BA (0.1b) databases. Note that this schema comes from the Hyperion project, see Chapter 6.

another, the flight row of the flight itself may be updated in AA's database, otherwise, a new entry is created.

fno_{AA}	fno_{BA}
AA210	BA1023
AA341	BA1078
AA341	BA1080

Figure 0.2: Example of mapping for data from AA to BA

In this case, BA selects all its flight data and pushes it on the network. Yet, in some other cases, all the flights may not be relevant. Let us imagine that AA only wants to know all the flights which destination is "New York". In this latter case, BA needs to select, in its local format, all the flights going to New York, and then push it on the network for AA to retrieve it and merge it in its database.
It is yet important to note that even though in the example above, the mappings are used to match two database structures, in our system, the mappings are used, as explained in Chapter 6, to match a local format to a universal one understood by all members of the sharing process. What happens to this sharing process when there are three or more members? In this case, each of them will have the data of everyone in its database and accepts that everyone, if granted, can modify this shared data. Thus, the data stored inside a member's database is obtained by merging the shared data of every member. Note that the network has only the purpose of permanently storing the data, which must be always available.

In this paper, we propose a user-friendly lightweight system based on bidirectional transformations (BX) [30] that can consistently manage the data sharing process. Indeed, the system provides a simple interface limiting the needed knowledge and enhancing its easiness. In this case, the user will neither be directly drown inside the complicated database nor be coding its way to share the data. The provided interactions can ease the handling of the system, where, once the data to share has been easily determined, the bidirectional functions take care of concretely sharing it in background. The following points summarize the three distinguishing features of our system.

- Our system, being based on BX's and coded in BiGUL [24, 29, 28], a bidirectional language, will automatically guarantee "well-behavedness", i.e. that ingoing and outgoing transformations are inverses. Indeed, bidirectionality is notably used to enhance the insurance of consistent propagation and update of the data thanks to its "well-behavedness" properties.

- Our system uses the authorization property provided by the bidirectional transformations themselves. With such a property, the system is able to authorize or not the propagation of updates according to the will of the users. The shared data is then either *writable* or *read-only*.
- Our system is extensible. Indeed, it has been designed in a way that a new user can join the group sharing data easily. Indeed, the user just needs to provide to the BiGUL functions a GLAV mapping table [27] matching his local format to the shared one. When this is done, the system uses the BiGUL functions to retrieve and later update (if authorized) the shared content.

Note that the system has been implemented [18] in Haskell and in BiGUL, a putback-based bidirectional language based on Haskell.

Chapter 1

Introduction to Bidirectional transformations

1.1 Bidirectional transformations

The bidirectional transformations (*BX's*) consist on a pair of transformations: a *forward* transformation and a *backward* transformation. They are used in many domains like relational databases [14], data synchronization, ... The objective is to maintain a system well-behaved. To do that, the BX has two different functions : the *GET* and the *PUT*. The *GET* is the forward transformation. This transformation will generally take a general system (called *source*) and produce a smaller system (called *view*). Intuitively, the *source* contains more information than the *view*. The backward transformation *put* take both (a *source* and a *view*) and will update the old *source* with the new *view*.

A bidirectional language helps in coding bidirectional programs. These programs will generate the *GET* and *PUT* with only one specification. There are two different types of bidirectional languages. The first types are the get-based. It will base the code on the *GET*. Then, the user will code the transformation of the source to produce the view. This kind of language is easier to understand and to use, but there is a major problem. For a given *GET*, there are multiple possible definitions of *PUT*. Get-based language needs to manage these multiple definitions to keep the consistency. Another solution is to use the second type of bidirectional language, the put-based language. It consists of coding the *PUT*. A put-based language will define the way of updating the old source with a new view. It is more difficult to code that transformation. But the big advantage is that *GET* comes for free and there is only one possibility for the *GET*.

To keep the consistency, bidirectional programming will use *lenses* . A *lens* is a bidirectional program that satisfies the well-behaved properties below. There are two different types of lenses : the symmetric [44] and the asymmetric lenses.

The symmetric lenses are made to keep the consistency at any time. To do that, the idea is that the *PUT* will also return a complement [44]. That new complement will be used as a second argument to *GET*.

$$
\begin{aligned}
get &\in X \times C_Y \rightarrow Y \times C_X \\
put &\in Y \times C_X \rightarrow X \times C_Y
\end{aligned}
$$

As mentioned in the symmetric lenses paper [44], *C_X is the "information from X that is discarded by get", and C_Y is the "information from Y that is discarded by put". We can think of the combined complement C as $C_X \times C_Y$ —that is, each complement contains some "private information from X" and some "private information from Y "*. In this situation, the get has been renamed to *putr* and put to *putl*. In this paper [44], a symmetric lens is defined like this : *A lens l from X to Y (written $l \in X \leftrightarrow Y$) has three parts: a set of complements C, a distinguished element missing $\in$ C, and two functions.*

$$\begin{aligned} putr &\in X \times C \rightarrow Y \times C \\ putl &\in Y \times C \rightarrow X \times C \end{aligned}$$

satisfying the following round-tripping laws:

$$\frac{putr(x, c) = (y, c')}{putl(y, c') = (x, c')} \quad \text{(PUTRL)}$$

$$\frac{putl(y, c) = (x, c')}{putr(x, c') = (y, c')} \quad \text{(PUTLR)}$$

Asymmetric lenses are more flexible. The *GET* and the *PUT* can be used separately. To prove that a asymmetric language is *well-behaved*, lenses will satisfy these two rules :

$$put\ s\ (get\ s) = s \quad \text{GETPUT}$$
$$get\ (put\ s\ v) = v \quad \text{PUTGET}$$

The first rule, GETPUT, requires that the *view* produced by the *GET* will not modify the *source*. The second rule, PUTGET, requires that the modified view can be recovered with the *GET* from the updated *source*. All programs that satisfy these rules are *lenses*.

1.2 BiGUL functions

BiGUL is an asymmetric put-based bidirectional programming language. This language give us some tools to code bidirectional programs [24]. BiGUL is also based on Haskell. A bidirectional program

$$bx :: BiGUL\ s\ v$$

is a way to describe manipulations between a *source (s)* and a *view (v)*. To describe the manipulation, BiGUL will produce two functions (*get* and *put*). The *get* function return a view from a *source* and the *put* function return a *source* updated by the *view*.

$$get\ bx :: s -> Maybe\ v \quad (1.1)$$
$$put\ bx :: s -> v -> Maybe\ s \quad (1.2)$$

The language offers tools to specify the link between the *source* and the *view*. BiGUL is an put-based language : it specifies the link with the *put* function. For example, when you replace a value, that means the *view* will update the *source* by replacing values. The *Maybe* will return *Just x* or *Nothing*. BiGUL has a possibility to trace the *get* and the *put* by calling *getTrace* and *putTrace*.

1.2.1 Skip

The first function is *Skip* (1.3). The idea is to keep the original *source* but also check the function.

$$Skip :: (s -> v) -> BiGUL\ s\ v \tag{1.3}$$

For example, if there is a simple Haskell function : *next x = x + 1*. With (Skip next) (1.3), you can preserve the *source*. The *put* function determine if the link between the *source* and the *view* is correct. In that case, it will return the unchanged *source*. In the other case, it will return *Nothing*.

```
put (Skip next) 10 11
Just 10

put (Skip next) 10 12
Nothing
```

The *get* function will apply the function skipped.

```
get (Skip next) 10
Just 11
```

1.2.2 Replace

The second function is *Replace* (1.4). The *source* is replaced by the *view*.

$$Replace :: BiGUL\ s\ s \tag{1.4}$$

Replace takes a *source* and a *view* with the same type to replace it. The *get* function will return the current source.

1.2.3 Rearrangement

What if the *source* and the *view* have the same structure ? BiGUL can change the structure before applying a function. The *source* (1.5) and the *view* (1.6) can be rearranged.

$$\$(rearrS[[e :: s1 -> s2]]) :: BiGUL\ s2\ v -> BiGUL\ s1\ v \tag{1.5}$$
$$\$(rearrV[[e :: v1 -> v2]]) :: BiGUL\ s\ v2 -> BiGUL\ s\ v1 \tag{1.6}$$

[[e :: x1 -> x2]] is the λ function to rearrange the element. For example, to apply a BiGUL function on just the head of the list, the rearrangement will be like that : *$(rearrS [[(x:xs) -> (x,xs)]])$*. Here, the rearrangement change a list into a couple with the head and the rest of the list. The rearrangement is here to help the user to modify the structure but these functions can't modify the values of this structure. Then , the wildcard ('_') is not allowed in rearrangement.

1.2.4 Prod

As it is mentioned on the rearrangement, the structure can change in a BiGUL program. In our example, a list will become a couple. In that case, The BiGUL function applied can take the whole structure or just one part of the structure. *Prod* is a function that allow the programmer to use more than two BiGUL functions. For example [24] :

```
pHead :: (Show a) => BiGUL [a] a
pHead = $(rearrS [| \(s : ss) -> (s, ss) |])$
        $(rearrV [| \v -> (v, ()) |])$
        Replace `Prod` (Skip const())
```

Listing 1.1: Some code with Prod

The BiGUL function *pHead* will *get* the head of the list and *put* the *view* in the head of the list. Here, it's two BiGUL functions into one. We will *Replace* the head and *Skip* the rest of the list. The *Prod* will run these two functions. The first BiGUL function (Replace) will be executed on the head of the *source*. *Prod* is typed as follows (1.7) :

$$Prod\ ::\ BiGUL\ s1\ v1\ ->\ BiGUL\ s2\ v2\ ->\ BiGUL\ (s1, s2)\ (v1, v2) \tag{1.7}$$

To Facilitate the reading of *Prod*, it can be write between the two parameters with enclosed air-quotes. *Prod* can be also use in more complex structure.

```
put (((Skip const ()) `Prod` Replace) `Prod` (Replace `Prod` Replace
   ))           ((1,5),(2,6)) ((() ,4),(2,5))
Just ((1,4),(2,5))
```

1.2.5 Compose

BiGUL also provides ways to compose bidirectional transformations using functional composition (1.8).

$$Compose\ ::\ BiGUL\ a\ u\ ->\ BiGUL\ u\ b\ ->\ BiGUL\ a\ b \tag{1.8}$$

Let's reuse the previous generic BiGUL function, *pHead*. Assume the source has the type list of lists *[[a]]*. The idea will be to create a function to update the head of the first list. To do that, *Compose* is the best solution because *pHead* is already defined.we define like in [24] :

```
pHead2::(Show a) => BiGUL [[a]] a
pHead2 = pHead `Compose` pHead
```

Listing 1.2: Some code with Compose

And then, as seen in the example below, only the head of the first list will be updated by the *view*.

```
put pHead2 [[1,2],[8,3],[5,8],[]] 20
Just [[20,2],[8,3],[5,8],[]]

get pHead2 [[1,2],[8,3],[5,8],[]]
Just 1
```

1.2.6 Case

Depending of the source and the view, it is possible that the programmer wants to execute different functions. The *Case* is here to identify all the cases and what the programmer wants to execute. There are two possibilities to characterize theses cases (*normal* and *adapative*).

```
Case [ $(normal [| mainCond :: s -> v -> Bool |] [| exitCond :: s -> Bool |])
       ==> (bx :: BiGUL s v)
     , ...
     , $(adaptive [| mainCond :: s -> v -> Bool |])
       ==> (f :: s -> v -> s)
     , ...
     ]
   :: BiGUL s v
```

With a *normal* case, there are two predicates (mainCond and exitCond). The *main condition* is just like a precondition. When the condition is met, the function in the case will be executed. The predicate for the main condition is a function that return a boolean value depending to the source and view. The *exit condition* is similar to a postcondition. After the execution of the bx function, the source will respect the exit condition. The predicate for the exit condition takes only the source.

For an *adaptive* case, there is just a *main condition*. When this condition is met, the adaptive case will execute a function to modify the source into a new one. The objective of adaptive cases is to transform the source to go back in a normal case. When an adaptive case is executed, the *Case* function is rerun. During this rerun, a normal case needs to be executed. if not the programmer will have a error.

Also, BiGUL provides some tools to help the programmer with the main condition. The idea is to separate the condition on the source and the condition on the view. Then there are two new definitions, one for the normal case and one for the adaptive case.

```
$(normalSV [| sourceCond :: s -> Bool |]
           [| viewCond :: v -> Bool |]
           [| exitCond :: s -> Bool |])
  ==> (bx :: BiGUL s v)

$(adaptiveSV [| sourceCond :: s -> Bool |]
             [| viewCond :: v -> Bool |])
  ==> (f :: s -> v -> s)
```

To code bidirectional transformations easier, there is the *emb* function. When the programmer coded the *get* and the *put* separately, the *emb* function will combine them to create a BiGUL function.

```
emb :: Eq v => (s -> v) -> (s -> v -> s) -> BiGUL s v
```

Most of the time, the *emb* function is used to define small and simple bidirectional transformations. For example, the source is a tuple with two integer and the view will be the result of the addition [24].

```
1 pSum :: BiGUL (Int, Int) Int
2 pSum = emb g p
3          where g (x,y) = x + y
4                p (x,y) v = (v - y,y)
```

Listing 1.3: Some code with emb

To be sure that the function is well-behaved, let's check if the GetPut and PutGet are respected.

```
get (fromJust (put pSum (10,5) 10))
Just 10

put (10,5) (fromJust(get pSum (10,5)))
Just 5
```

1.2.7 Align

Finally, there is the *align* function. This function allows an alignment of the source and the view. It is only possible if the source and the view are represent by lists. The idea is to be able to synchronize these lists. The *align* function is able to insert and delete new elements and keeps the consistency. These two lists don't need to be sorted before using the *align* function.

As you can see below (listing 1.4), the signature of *align* is complex. It needs 5 different functions to work.

```
1 align :: forall s v (Show s, Show v)
2          => (s -> Bool)
3          -> (s -> v -> Bool)
4          -> BiGUL s v
5          -> (v -> s)
6          -> (s -> Maybe s)
7          -> BiGUL [s] [v]
```

Listing 1.4: align signature

To explain the 5 functions, we choose to use an simple example (Listing 1.5). In this example, we want to *GET* some elements of the list and *PUT* the changes in the source.

```
1  alignString :: BiGUL [(Bool,String)] [String]
2  alignString = align
3          -- not deleted item
4          (\(b,_) -> b)
5          --match between the source and the view
6          (\(b,s) v -> ( s == v )
7          --update function
8          $(update [p| (_,x) |] [p| x |] [d| x=Replace |])
9          --create a view from a source
10         (\v -> (True, v))
11         --delete an item from the source list
12         (\(b,s) -> Just (False,s))
```

Listing 1.5: align example

The first function is here to define if we use or not an item in the function. In the example, all item with *True* in the tuple will be aligned. The second function is condition to match between the source and the view. Here, we will just consider that the source is equal to the view. If this condition is satisfied, the third function is used. This function is always a BiGUL function. In our example, this is a simple *Replace* like defined earlier. The fourth function is only use for inserting a new item in the source. The idea is to create a source with a view. The last one is the way to delete an item in the source. Depending of the first function, the last function is the transition between a regular item and a deleted item.

Chapter 2

Preliminaries and related work

This chapter sets a brief state of the art by giving a sketch of the SQL Views synchronization and the "Foreign Data Wrapper" (FDW) [23] but also of the different previous works from which this system is inspired. Those related works are namely "PeerCSDB"[41], "The Hyperion Project"[12], "ORCHESTRA"[21] and "Relational Lenses" [14]. It is important to note that not very aspect of those are relevant in this current system, but in order to understand the purpose of those projects, a whole but generic description is given. As most of these do not use bidirectional transformations, a section stating why the BX approach is interesting in the database synchronization perspective.

This chapter is organized as follows. First, the SQL Views synchronization is explained. Then, so is the foreign data wrapper. Third, the related works are described in this order after a brief reminder on peer-to-peer networks : "PeerCSDB", "The Hyperion Project", "ORCHESTRA" and "Relational Lenses". Finally, the importance of BX in the system and more generally in the database synchronization process is explained.

2.1 SQL Views Synchronization

This section states how the database synchronization [33, 8, 9] is currently dealt with in the case of different and similar database structures. This state of the art is probably not exhaustive, but it gives an overview of the actual solutions provided to answer the database synchronization problem. After explaining the four different synchronization methods and recognizing the pros and cons of each, the general disadvantages of those methods are given.

2.1.1 Structures and synchronization methods

As mentioned above, the synchronization between two databases can happen either when the structure is the same or very similar or when it is completely different. Yet, in this last case, most of the methods explained below are difficult to apply. Nevertheless, it still depends on the exact problem to be solved and its complexity.

2.1.1.1 Import data

When the structure of the databases are different, the most common method is to simply import the first database into the second. This process can obviously be automated and repeated, so that the data could be "synchronized" after a certain amount of time or after any other event set by the administrator. This method avoids the structure matching problem by copying directly a database into another. It can be thought as a "dirty" but it still remains effective.

2.1.1.2 Data comparison

Another synchronization method is data comparison, used when the structures are similar. This approach requires a tool to compare the data from the source database to the data of the target and to generate SQL queries containing the updates to be done.

As the structures have to be similar but not exactly the same, some adaptations are possible as, for example, matching tables on another constraint than their name, or changing a NULL value into an empty string. Also, by default, the tools use UNIQUE or primary key constraint for identifier. Yet, most of the tools allow the user to set the column or columns having that role. Note that among popular tools, it is possible to find "dbForge Data Compare for SQL Server" [7], "RedGate SQL Data Compare" [5] and "Apex SQL Data Diff" [1].

The advantage of such method is notably that the user must not have a deep SQL knowledge as the whole synchronization process can be done within the graphic interface. Also, this GUI gives the opportunity to the user to visually see the effectiveness of the changes operated.

A major drawback of this approach is that the generated requests cannot be reused from a synchronization to another as it is data dependent. The user is then obligated to use the advanced commercial tool every time he wants to synchronize the data. Finally, when a huge amount of data needs to be synchronized, the performance significantly decreases compared to other methods.

2.1.1.3 Automatically generated SQL scripts

This method is in a way quite similar to the data comparison. Indeed, the user uses a GUI to select the source and target databases, setup the different options and adaptations and finally launch the synchronization. Yet, the critical difference is that those automatically generated scripts do not contain any data, but only the synchronization logic. So, the scripts can be reused at will and a synchronization can be planned on the server as a file containing the requests can be loaded. This is a major advantage to be added to the ones of the data comparison method. Another advantage is the increased performance compared to the previous method.

There are yet two drawbacks. The first one is that the tool is an advanced commercial software, sometimes hard to handle. The second one, more technical, is that it is not possible to distinguish the differences between databases as no data is taken into account.

Note that the only tool known to be working with this approach is "SQL Database Studio" [6].

2.1.1.4 Manually created SQL scripts

The last method is to manually create the synchronization scripts. Compared to the comparison and auto-generation methods, it has the advantage to dispose of open source tools too. Otherwise, as this method uses a script, it also can be executed following planned interventions on the server as a script can be loaded. So, the server can automate the synchronization using different script, each of them having a particular task to complete among *update*, *insert* and *delete*.

Even though the developer is able to parametrize entirely the required primary and foreign keys, mappings and so as he like, this method has serious drawbacks. Indeed, the person behind those script may not be a user anymore but a developer. This creation may effectively require much SQL knowledge, which is not common to everyone. More than knowledge, the tediousness of the scripts creation is to be considered. In general, for a single table, three scripts respectively dealing with *insert*,*update* and *delete* must be created. When the structure of the table or view changes, the related scripts must of course be maintained, which creates an additional cost. Finally, this method only considers SQL queries, so it is not possible to import for example XML files or CSV files, as far as we know.

2.1.2 Drawbacks of the synchronization methods

Even though those methods provide relevant and efficient synchronizing techniques, it still has a very critical limitations.

Indeed, in all methods, the implication is that the user has access to both databases, which is not always the case, mainly in the example given in the Introduction. When companies want to share and synchronize data, they do not want to effectively give an access to their database to another company.

Another limitation of those techniques is that, as far as we know, the tools only provide a synchronization between two different databases. If many databases need to be synchronized, a solution needs to be found only by working with two databases at a time. It is obvious that, in the case of a huge amount of companies being members of the sharing process, those methods may not be so efficient anymore.

2.2 Foreign Data Wrapper

A foreign data wrapper (FDW) [23] (SQL2003) is another technique to directly exchange and synchronize data between databases. From a general perspective, a foreign data wrapper allows a database user to create a database interfacing a remote database that he does not possess. So, when a query is executed on the interfacing database, the query is automatically propagated on the remote one and sent back in the interfacing database so that the user feels like he is directly dealing with the remote database.

The goal is to use a shared server containing connection information used by members of the sharing process. This information is used to connect to the database to be shared. Another person having the address of the server and proper credentials is then able to access the shared database and make it match to its own. This technique allows then to have multiple databases using the server to interface the remote database. It then provides the starting point of a solution to the multiple database synchronization problem.

To use the FDW, the user of a PostgreSQL database, for example, needs to first, install the plugin. Then, he needs to create a foreign server object representing the databases that he wants to share. After, the developer needs to set the user mapping by giving the credentials of every user allowed to access the remote server. Finally, he needs to import the foreign schema or build himself a schema matching the remote database. The names of tables and columns must match, even if they can be aliased in the interface.
As it is possible to understand, this technique consists in coding and parameterizing a foreign server to interface the remote database in order to match formats. This method is not possible for users who are not developers, which is a huge limitation.

2.3 Problem to be solved

Knowing different SQL approaches to solve the database synchronization problem, it is now possible to answer the following question : what is the problem that this system is trying to solve? It is yet necessary to know that the first goal of this thesis is to concretely develop a system using BiGUL. Then, concerning the database synchronization, the system tries to address some of its issues, namely the knowledge of both databases structure and the extensibility of the sharing process. When trying to solve those issues, the system still needs to keep the advantages of some existing approaches, such as the easiness of the user interface and the automation of the synchronization. This system also answers directly to the data consistency issue relative to every sharing process by using bidirectional transformations enhancing this consistency.

In a more technical perspective, the goal is to synchronize the data from a database by inserting it inside an existing table in another database. Let us take the example of the Introduction to illustrate. There are two airlines, A and B, related in some way. Company A wants to share flight data with B. As the goal is the merge A's data inside B's table and vice-versa, they need to know the each other's format. Yet, to avoid such constraint and thus reduce the necessary knowledge but also increase the extensibility of the system, mappings are elaborated to translate the local format into a universal one understood by everyone. Company A has then only to know its local data and its matching to the universal format. Using only the interface of the system, the user is then able to push the *read-only* or *writable* data on the network. Company B is then able, using the same system interface, to retrieve the data and, after matching it to its local table, use it in any way they want, as it is in their database.

Concerning the advantages of the automatically generated SQL scripts, in the current implemented version of the system, the user can only partially automatize the synchronization if the data that he wants to share respects the same conditions (i.e. if the selection condition and the columns remain the same, whatever may the selected rows be). However, because of some communication constraints (see Chapter 7), the data retrieval cannot be automatized. Yet, the new version of the communication protocol proposed can easily remedy that problem, allowing a developer to create a small script to automatize the retrieval if he wants. It is important to note that by "automatizing", it is meant that the user will not have to execute the system himself. Also note that even if creating scripts to automatically execute the system would be possible, it has not been considered in this thesis.

The following sections describe whole data sharing systems that have been thought about in the past and on which this system is based.

2.4 Peer-to-peer networks

As most the related works use a peer-to-peer (P2P) communication system, it is important to remind the bases of those networks. This section sets then these bases, describing the different application types and infrastructure models. The Chord protocol used in this system is also briefly introduced in this section.

2.4.1 Application types

Even though most people know P2P networks by file sharing systems such as *BitTorrent* [13], peer-to-peer networks are more complete and complex than that. They not only provide a way to share files but also and mainly "shared resources in order to provide a

service that the system has been designed to provide" [40]. To be more exact, the peers, or nodes of the network, can request and/or provide services to other peers. They will act at some point both like a "web server" and a "web client", as it is for example the case in file sharing systems. So, it is possible to find four kinds of applications : content distribution, distributed computing, collaboration and communication platforms like JXTA [11].

The content distribution applications are made exclusively for file sharing, like Gnutella [39], BitTorrent, etc. Distributed computing applications are used to have parallelism as well as grid and cloud computing [35]. Collaboration consists of instant messaging, like P2PSIP [15], while platforms like JXTA are responsible for peer discovery, grouping of peers and communications between peers.

2.4.2 Infrastructure models

There are three different models of infrastructure : centralized directory, decentralized directory and fully decentralized directory.

As shown in Figure 2.1, the *centralized* model contains a master server which indexes both information about every peer that connects to the network - mainly its IP address - and the shared content. If another peer wants to retrieve that content, it has first to contact the master server to be able to reach the peers storing the data or a part of it. A good example of such a model is the *Napster* system.

Yet, this infrastructure shows some major drawbacks, principally on the main server. It is indeed a single point of failure (SPOF) as every peer must use it to locate the content. So, even if the file transfer is decentralized (the peers retrieve the content directly from each other), the lookup is not, decreasing the robustness of the system in some cases, notably when the load is very high. The server is then not only a SPOF, but also the bottleneck of the infrastructure.

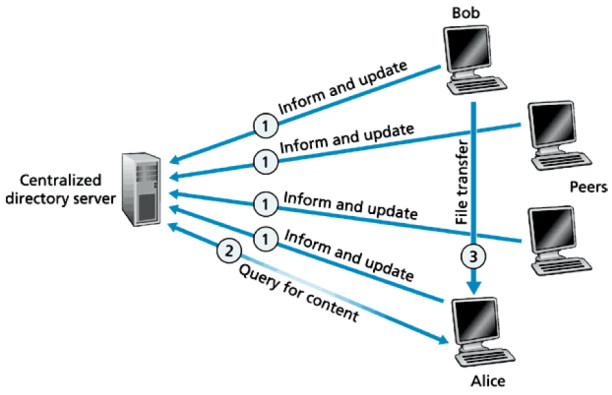

Figure 2.1: The centralized directory model

The *fully decentralized* infrastructure proposes the opposite. In this case, every peer has indeed the same importance as none has a central or leading role in the system. The concerns raised by the centralized model are then answered, yet, some others arise. For example, in the case of a very large network, as every peer has the same role and because it is impossible for each of them to know the network layout, looking up for data or a peer might take a while and many queries. Even though, thanks to protocols like CHORD [42],

the network remains scalable. The cost of such a scale is just increased.

In the absence of a lookup protocol, as shown in Figure 2.2, to find data or another peer on the network, the sender needs to flood the network in every direction. The peer concerned then answers directly to the sender when the query is received. In the case the query does not reach the desired peer or data, a timeout is needed in the sender's system, so that it does not wait indefinitely.

The *decentralized* directory model or hybrid peer-to-peer network is, as its name implies, a combination of the two models described above. This infrastructure differentiates the leader nodes and the "common" nodes. Each leader acts as a central server for the common peers under its responsibility. If this leader does not possess the information queried by one of its children nodes, it uses the overlay network (fully decentralized infrastructure) to connect another leader node in order to retrieve the adequate data. It is now understandable that this model can be decomposed into the two infrastructures explained before, as shown in Figure 2.3.

Depending on the implementation, a leader can be either elected or voluntarily propose to endorse this role. It can be seen as a major advantage, because if a leader fails or leaves, another peer with enough computation power can take its place, making the whole network quite hard to completely shut down. Also, as the leader is considered as a central server by its children, the drawbacks and advantages of the centralized model apply here. For example, if there is a heavy load on this node it can be overloaded and become a bottleneck. It is also important to notice that this decentralized model naturally inherits the query flooding infrastructure properties.

2.5 PeerCSDB

Knowing the basics of P2P networks, it is now possible to introduce previous works for peer-to-peer database sharing/management systems. In order to understand PeerCSDB - being one of those projects - [41], an overview of its infrastructure is needed.

Unlike other peer-to-peer database systems, PeerCSDB consists of a composition of two different architectures. It indeed takes advantage from both Client-Server and peer-to-peer models, as a main server containing the database, and clients being also peers in the P2P network are both in the system. As all the shared information is stored into the database of

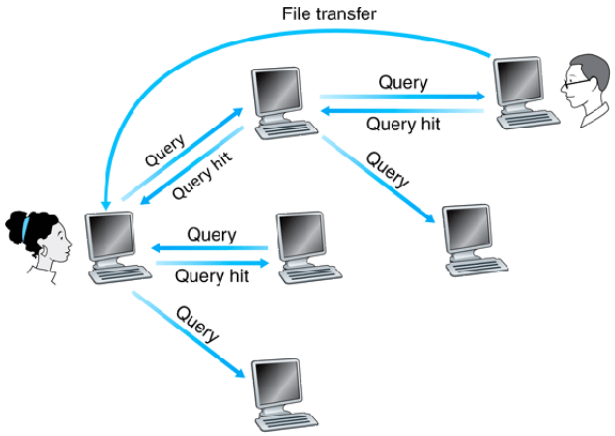

Figure 2.2: The fully decentralized directory model

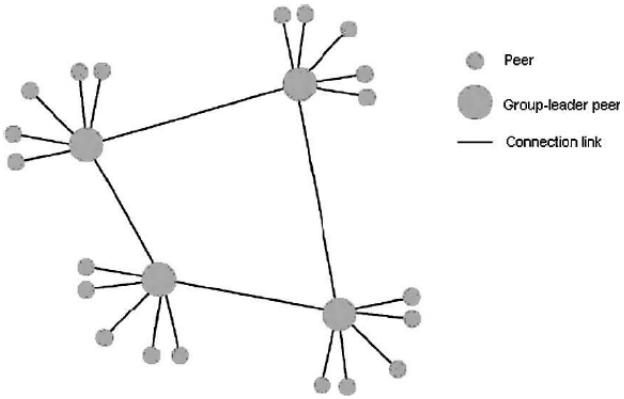

Figure 2.3: The decentralized directory model or hybrid peer-to-peer network

the main server, all the peers would need to emit queries in order to get the data they want to retrieve. This description can be recognized as the Client/Server architecture, where the server can be the bottleneck in case of a large amount of clients. In order to avoid such problems, the set of peers has been divided into two parts or layers as shown in Figure 2.4.

2.5.1 Resource sharing layer

The first layer, named "resource sharing layer" contains peers that will cache chunks of data from the main server in order to avoid queries on databases when possible. But if those chunks are stored into a database on the peers, the access time will not significantly be reduced, due to database access and lookup costs, notably in time. The data needs then to be cached in different way than database techniques. XML files are then chosen, as it is easily possible to convert a database into such kind of file, and as a XML document is only a structured text file which can be read easily. Yet, if a whole table is stocked inside a file and if a query only matches few rows, the whole table would have to be checked, which is not efficient. Therefore, only the query results will be saved in the data cache. In that case, if a query has already been done once, the peer will answer faster. If the node cannot answer the query, it will then query the main server by itself, retrieve and cache the query result and then send it back to the peer which asked for the data in the first time.

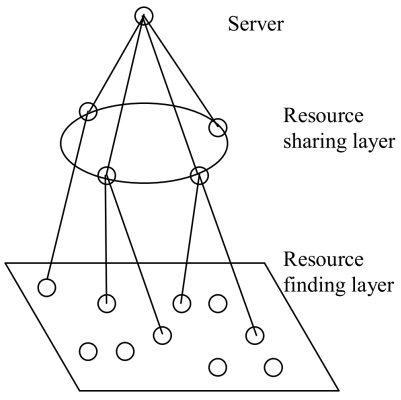

Figure 2.4: The architecture of PeerCSDB

Even if ninety-nine percent of queries are "select", it might also happen that a modification is made by a peer on the data. If so, the whole data cache about the table where modifications happened will become invalid. That is why a new cache containing the primary keys and

the query conditions is created. After the modification of a record, the whole primary key cache will be checked, setting the couple of primary key and query condition to invalid if the query condition matches the update. If a certain amount of "invalid" is reached, the cache will be renewed by querying the main database.
It also is important to note that all the data is not replicated inside one peer, but they will share equally - or almost - most of the chunks of data present in the database. The peers on the resource sharing layer are all on the same network managed by the CHORD protocol.

2.5.2 Resource finding layer

The second layer, the "resource finding layer" contains only peers that want to retrieve data and send queries to the resource sharing layer. Those peers do not cache any data from the database and use a query flooding network to connect among themselves. The resource in this layer is not the data to be shared but the IP addresses of the peers in the resource sharing layer. If a node willing to retrieve data does not know any IP address of any peer caching data, it just needs to ask its neighbors, and wait until one of them answers or until the TTL is reached.

2.5.3 Aspects of PeerCSDB

The architecture of this project actually matches well with a decentralized directory network infrastructure. Indeed, this two-layer architecture is useful to efficiently propagate data modifications on the whole network while avoiding to loose the advantage of structured networks when the number of peers is huge. Following this comparison logic, the peers on the resource sharing layer, called super peers, have the leader role defined in Section 2.4.2 while peers on the resource finding layer are "ordinary" peers. As in the hybrid model, "ordinary" peers can, if they have enough computation power, voluntarily propose to become a super peer.

Even though the style of infrastructure matches well, it is important to notice that the networks between peers of the same kind are inverted. Indeed, in this project, super peers and "ordinary" nodes respectively use structured p2p and unstructured networks, while in the decentralized model, leaders and "ordinary" nodes respectively use overlay network and structured p2p networks to communicate among themselves. In this project, the structures use different protocols, namely Chord for the resource sharing layer and Gnutella for the resource finding layer.

Even if PeerCSDB does not provide a complete answer to the problem presented in this thesis, some of its aspects are still relevant. This is notably the case for the peer-to-peer communication protocol in the resource sharing layer (i.e. CHORD), which is used in the current project. Moreover, the selection and check of primary keys to validate updates is relevant too.

2.6 The Hyperion project

If the PeerCSDB system has the only purpose of sharing the data of a single server, the Hyperion project [12] proposes to access the data stored in each node via peer-to-peer network. In that case, each peer has its own data which can be remotely accessed in order to exchange information. In this case, the fetch mechanism is the major part of the system. Also, every node on the P2P network is a PDBMS, abbreviation standing for *peer database management system.*

2.6.1 Mappings

As mentioned previously, each peer possesses its own data in its own format. Yet, all acquaintances are able to access others' data and exchange their own. To do so, Hyperion uses *mapping expressions* and *mapping tables*. A mapping expression is a mathematical relation between the data of two peers. For example, if company A owns company B, the data of company B will most likely be a subset of company A's data. Such an expression does not limit or express any constraint on the contents of the peers itself, but on the relations between them and on the place where to fetch some data. Let us take the same example as above, where company B is owned by company A. Requests on company A can then include information concerning B that are stored in B's database. That's why the mapping expression is important to know if another peer is a subset of the current peer. The part of the request concerning B is then directly sent to and processed by B.

To complete those expressions, mapping tables have been created. They map a value inside a peer's database to another value inside another peer's database, as the formats are most likely different. Using both mapping expressions and tables, the queries can then be processed from a peer to another and converted into a format understandable by the queried peer.

2.6.2 The managers layer

Another important aspect of the Hyperion project consists of the implementation of the P2P layer, being the communication and query manager layer. Specific protocols have been created to match the needs of the system.

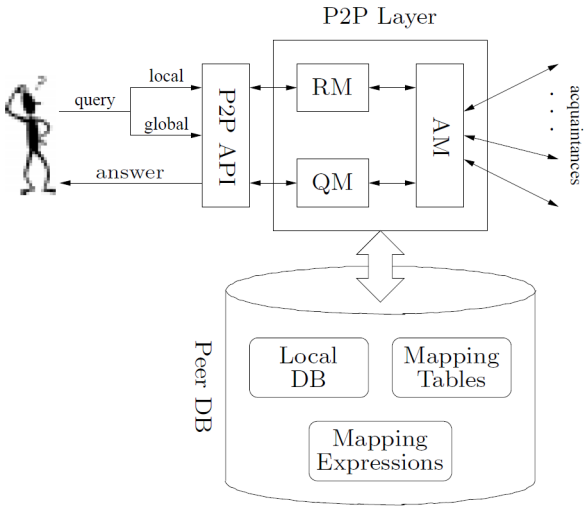

Figure 2.5: The architecture of a peer (a PDBMS) in the Hyperion project

As shown in Figure 2.5, each peer is composed of three major parts : the P2P API, the P2P Layer and a DBMS. While the DBMS contains all the data including mapping tables and expressions, the P2P API is used as an interface between the user and the system. The P2P Layer is a critical part of the system, mainly because of its composition. Indeed, as described in Figure 2.5, it is composed of the rule manager (RM), the query manager (QM) and the acquaintance manager (AM).

2.6.2.1 The rule manager

The goal of the rule manager is to maintain consistency between peers, using event-condition-action (ECA) rules performed through the API. Those rules, based on mappings, ensure that, if the data on one peer is modified, the data on the other peers will also be updated. They can be understood as some kind of SQL triggers, as written in [12].

2.6.2.2 The acquaintance manager

The objective of the acquaintance manager is, as its name indicates, to be responsible for PDBMS acquaintances. As two PDBMSs are really linked to each other when mapping tables and expressions exist between them, the role of the AM is to create and maintain those mappings. They are created by both user input and automated processes, making the creation process semi-automatic. The system being dynamic, the mapping tables and expressions are too. In that case, the AM needs to correctly and consistently maintain links and data between acquaintances.

2.6.2.3 The query manager

In order to explain the role of the query manager, it is important to distinguish two different types of queries : local and global. The local ones are only executed inside a single PDBMS, asking only for the data which is locally present in the peer. The global queries not only retrieve data in the peer, but also data in other peers. These queries need then to be propagated among acquaintances. The query manager then transforms, with the help of the acquaintance manager retrieving mappings, the global queries into a format understandable by each other peer to be queried.

2.6.3 From the Hyperion project

From this project can be extracted two major ideas, namely mappings and peer-to-peer communication. Indeed, the idea of mapping is very interesting in the context of database synchronization in order for peers to know only their own schema and then to ease the task of the user. Also, in that case, peers do not have to know the whole network, but only their own data and acquaintances, which is enough to keep the whole system working. Yet, the peer-to-peer communication is not based on a preexisting protocol, as the p2p layer and API have been specifically designed and in summary, Hyperion follows a peer-to-peer philosophy, without centralization.

2.7 ORCHESTRA

As the Hyperion project, the purpose of ORCHESTRA [21] is to share the data present in each peer and propagate it among the network. Also, it will use mappings as *tuple-generating dependencies* (tgds)[19], which are similar to GLAV (*global-local-as-view*) mappings [27] used to map a database schema to another. As the peers will completely share and send data among each other, not only "fetch and propagate updates" like in Hyperion, the system is called a CDSS, namely *collaborative data sharing system*.

Moreover, as data from other peers resides inside one's database, the system proposes trust policies. This allows a peer to decide whether to let or not another peer update its content based on trust conditions. Yet, in order to do so, it is needed to know where the data comes from, leading to the creation of provenance information carried in updates. Thanks to the provenance, if an update is accepted, the peer in question will propagate this update to its acquaintances in order to keep the data consistent.

2.8 Relational Lenses

This section describes in a nutshell the "relational lenses" [14], a language for updatable views based on the Harmony project [37]. This previous project actually proposes lenses to solve a generic view update problem, as long as the data structure is a tree, such as an XML file. Even though relational data can be converted in trees, dealing with the functional dependencies carried with this data is a problem for Harmony. That's why the "relational lenses" language has been created, based on relational algebra.

As it was the case for the Harmony project, the relational lenses language is also necessarily "well-behaved" in the sense defined in Section 1.1. Yet, this relational approach is said in [14] to be completely new, by using bidirectionality to solve the view update problem. It contains three major primitives : *join*, *drop* and *select*. They are all "well-behaved" relational lenses. For each expression formed by combining the primitives, a view definition must be given as well as a view update policy.

This work is then very related to ours. Indeed, both try to solve the database sharing problem by addressing the update problem itself. Yet, a huge part of this work is about proving the well-behavedness of the different lenses proposed to solve the update problem. Also, the "relational lenses" is just a language used to synchronize databases in both *get* and *putback* ways. While this work proposes a new language to be used for database synchronization using lenses, the purpose of our system is quite different. As explained in Section 2.3, it is meant to be used by users, not developers, to provide a solution to the database synchronization problem. To resume, the "relational lenses" bring a new approach to the database synchronization problem, approach that our system uses, but in a more practical and easier way.

2.9 Bidirectional transformations in the system

As shown above, many different approaches have been thought of to solve the database synchronization problem, most of them not using bidirectional transformations. Yet, the use of BX is not new to our system, as the relational lenses language proposed that approach at first. Still, it seems like the bidirectional transformations are not practically used to solve such database synchronization problems. Knowing all this, what was our motivation to use BX to share content from a database to another?

The biggest reason behind the usage of bidirectional transformations in the system is the "well-behavedness" property that they have. Indeed, every bidirectional transformation coded in BiGUL is, as mentioned in Chapter 1, always "well-behaved", making the data processed always consistent. This property is really important for database synchronization where every data edition needs to be propagated in a consistent way in order to maintain coherence among all databases. Thus, with a language like BiGUL, well-behavedness comes for free by using bidirectional transformations of the data. Nevertheless, as disclaimer and following [28], the behavioral correctness of the system is not directly implied from the code. The behavioral correctness is the property proving that with given inputs, a program can produce outputs and that those outputs are the expected ones. In another words, it intends to prove on which inputs can the system produce outputs and whether those outputs are the expected ones. This can be obtained thanks to Hoare logic. Yet, as this kind of proof requires significant efforts, in this work, some tests have been used to give confidence in this correctness.

Another advantage of BiGUL is that it easily manages sharing privileges directly inside the code. This enhances consistency and data safety during the whole sharing process. Furthermore, the selection of data is neither a problem. Indeed, a BX coded in BiGUL can easily manage selection on lists, notably by using the *align* function or otherwise by easily setting conditions still maintaining the source and the view consistent.

Moreover, BiGUL may be lighter to code thanks to the *validity of PUT* and the *uniqueness of GET*. Thanks to those two characteristics described in Chapter 1, the programmer just needs, in most cases, to code the PUT direction, the GET one coming for free.

It is thus easy to understand why bidirectional transformations are the major part of this system and why it is also very useful in the sharing world, whether it consists of databases or not.

Chapter 3

Overview of the project

This chapter lays the foundation of the complete system, explaining its four different consecutive parts. This sketch of the system is done in a top-down approach, easier to understand.

3.1 Contributions to solve the problem

After defining the state of the art and introducing the transversal concepts, it is important to remember the major contributions to solve the problem stated in the previous Chapter. First, the system is quite simple to use by not requiring specific SQL code or knowledge, even though the interface is only developed in command line. This notably answers the difficulties of "the manually created SQL scripts" approach explained in Chapter 2. The system can be considered as a "black-box" system where the user's role is not the programmer's one, and where the interface is simple to use. Yet, some usability tests could be run to prove this affirmation.

Second, the use of bidirectional transformations greatly improves the data consistency, ensuring that if an update has been made on any view, it will be propagated back on the source. The "well-behavedness" of the bidirectional transformations is not to be proven anymore[1]. Also, BX's can easily check if updates are legitimate and authorize them or not, following different policies. In that case, the user can set a preference to share its data in read-only or writable mode.

Finally, the system is extensible. Indeed, any user can easily join the sharing network, even when data has already been shared. The user just needs to provide the GLAV [27] mapping and get the keys to pull the data from the network.
If any of these contributions description seems unclear, note that they all will be explained in details in the next chapters.

3.2 Overview of the system

In order to illustrate the contributions, an overview is given following both a user and technical perspective. While the first one is more a front-end description, the second perspective consists more in a general explanation of the system.

[1] Note that the correctness of the transformation still needs to be proven, by the Hoare logic or through experiments.

3.2.1 User perspective of the system

As mentioned above, one of the advantages of the system is to keep the interface simple to use for the user. Indeed, he is only asked very basic but necessary information, as in Figure 3.1.

```
Which Database do you want to use ?
bigul
Push or pull data ?
push
Which Table do you want to use ? (':q' to quit)
["a_b","album","album_tracks_join","bformat","test","tracks"]
album
Select columns you want to synchronize
["Album","Quantity","Rating"]
Album Quantity
Selection column ?
["Album","Quantity"]
Album
Condition ? (For example : '==10') ('*' for all)
==Show
Writable (T/F) ?
T
Do you want to synchronize this data (Y/N) ?
TableSync "album" ["Album","Quantity"] [([SqlByteString "Show",SqlInt32 3],True)]
```

Figure 3.1: The user interface with an example of data sharing.

When the system is used for the first time, the user needs to set it up, using the *setup* command. The main purpose is here to create the mapping table inside the user's database, among all other tables, so that every mapping will be stored there. To do so, the user needs to specify the database to be used, as a server can count many. Note that if not already existing, this mapping table is created thanks to the default values defined in the configuration file which can be edited by the user.

After configuration if needed, the user can call the *main* function to start sharing data with others. The system then asks which database to use. Indeed, the user my have set many databases up during the configuration part in order to share their respective content. After entering the name of the database, the user is then asked if he either wants to push or pull the data on/from the network. Only when the required information is given, the system connects to the database using credentials found in the configuration file. More concretely, credentials consists of a user ID, a password and the location of the database server, which is its concrete IP address and port. Again, if those happen to change, the user can edit the configuration file.
The purpose of this connection is to retrieve the structure of the database, more concretely its tables.
The user is then asked to select a table from the displayed list. Note that there is no restriction on the table to use, even the structure tables could be shared. After the tables, the user is requested to give the names of the columns to select, also from the displayed list, as shown in Figure 3.1. Following this selection, the user can now enter the column to which the selection condition is applied. Then, the condition itself needs to be given to the system, in the simplest way. For the moment, the system can only deal with equal, but has been implemented in a way that all conditions on every column type could be given. Even though the condition is really simplified, an example is still given in order to give the user and idea of the expected syntax.

When all selection is done, the data needs to be set either to read-only or to writable. To do so, the user just needs to enter respectively F or T. The system then does background computations using bidirectional transformations to display the data to be shared and asks for confirmation. The user can then see the concrete data in its exact syntax in which it is going to be shared with other people. Obviously, only the data to be shared is displayed. If the user agrees to share the data, the system then displays some information related to the network and pushes the data on it. Obviously, the system displays the list of keys needed to retrieve the data. After displaying a message confirming that the data has been successfully shared, the selection of tables are displayed again for the user to push another table on the network. To quit, *:q* can be typed at any step after the list of tables is displayed.

It is also possible that the system asks for correspondence between the universal and local format. Indeed, in order to give a meaning to the table name and columns, the system is not allowed to automatically generate random identifiers. The local names are therefore successively displayed and matched to a universal name chosen by the user. It is indeed important for those to be meaningful to ease the matching work of the user receiving data.

The description above is about pushing data on the network. Pulling it works in a similar way at the beginning, by selecting the database to use, in order to know where the changes need to be reflected back in case of modification on the data. The user then needs to type *pull* instead of *push*. The system then asks the list of keys representing the data on the network. As displayed in the command line, the order of those keys matters as they are retrieved and concatenated to have a meaningful message. For more information, please refer to Chapter 7. When the data is retrieved, it is possible that the universal format has no local equivalent yet. The user is then asked to type the name of the preexisting local column, table or primary key matching the universal one.
Finally, the system tells the user that the data has successfully been retrieved and inserted inside the specified database.

3.2.2 Technical overview of the system

As said earlier, this part consists of a more technical overview explained in a top-down approach. By "technical", it is meant background computations made by the system in order to complete the whole sharing process shown in Figure 3.2. It is also normal to find similarities between the user perspective and the technical overview, as it uses the same implementation logic.

As previously explained, to share data, the system has first to retrieve this data from the database by asking the user to give input interactive. He notably has to give the name of the database to use and the name of the interesting table(s).

Then, for each retrieved table, the user is asked which content he exactly wishes to share, selecting the rows and columns. Moreover, a major advantage of the system is that the user can choose which privileges he gives to others concerned by this data sharing. He can for example set the content to *"read-only"*, where all others will only be able to retrieve the data, or to *"writable"*, where everyone can update it. Yet, the edition privileges are defined for a whole sharing group. It means that the authorizations cannot vary from a sharing member to another among the same group. Anyway, it is possible to say that the system controls, thanks to bidirectional transformations, the authorizations to modify the shared content.

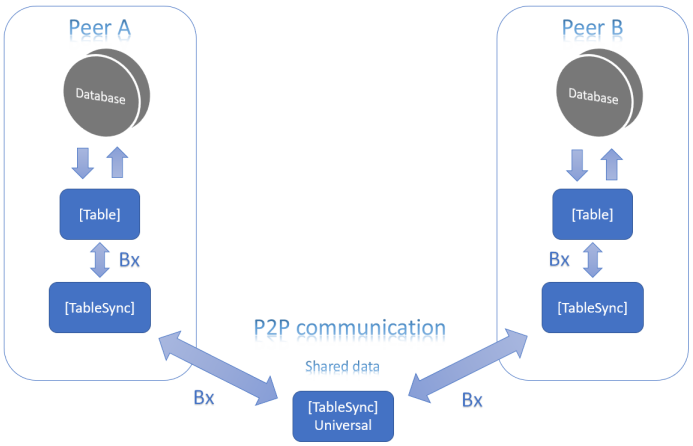

Figure 3.2: Global schema of the system.

After the selection, the tables are matched to a *"universal format"* defined by users when they share the data of the tables for the first time. This provides a way for the user to only know what is his data structure and what is happening inside of it, without even knowing a single column of the others' database. By using this *"universal format"* via mapping tables and bidirectional transformations, the guarantee of consistency is still preserved and the system is greatly simplified for the user.

When the content has been converted to the universal format, it can be sent on the peer-to-peer network, used mainly for the time-shared storage property. Indeed, those networks allow to join and leave at will, but also to always keep the data available. In that case, everyone concerned by the data sharing can retrieve and update the data at will, even if its owner is not online. As mentioned in the preliminaries, the protocol used for the p2p network is called *Chord*. The data on the network is then represented by hashed keys provided by and to the users.

This description only shows the way to share data, but neither how to receive it nor its updates and put it back in the database. Most of the system is coded using BiGUL, the "BX" arrows in Figure 3.2, so functions are meant to do round-trip modifications. In other words, most functions, even though only one implementation exists, can be used for both sending and receiving data. In this case, the procedure is applied in the exact opposite order, from the network propagation to the insertion or update of the database, passing by, by order, the matching of *"universal format"* to the local one and the reconciliation of shared data to the tables, as Figure 3.2 suggests.
When N peers use the system to share data, it is important to know that the system itself is exactly the same. The users just have different databases in various formats, to which the system extracts and merges the shared data. The system has then a direct access to the database and can directly modify its content, as the two arrows of the first "layer" between *Database* and *[Table]* show in Figure 3.2. In this Figure, the *P2P Communication* itself does not use bidirectional transformations, but the translation of the *[TableSync]* to or from the *[TableSync] Universal* does.

Chapter 4

Database to Haskell type

This chapter describes, with more details, the first step of the system. This step consists of a transition between the database and the Haskell type. Also, this step is the only part of the system without bidirectional programming. Just native Haskell has been used to code this part.

The main idea is to be able to retrieve the data but also to propagate the changes back to the database when an update has been effectively done and is allowed. To get the needed data from the server, the user needs to edit the configuration file provided with the system and notably listing all access information. The configuration file is a simply Haskell file (Config.hs). As you can see on the figure 4.1, an user can easily change the variables of the system. It is important to notice that the whole database is not retrieved, but only the content to be shared. This file is a personal file for every user. All the parameters for a connection to database are defined here.

```
1 mySqlhost :: String
2 mySqlhost = "127.0.0.1"
3
4 mySqlUser :: String
5 mySqlUser = "root"
6
7 mySqlPwd :: String
8 mySqlPwd = ""
```

Listing 4.1: configuration file example

4.1 Extraction of the data from the database

There are two problems to extract data from a database. The first one is to retrieve data from SQL data into Haskell. The second is how can we represent the data in Haskell type.

4.1.1 Package HDBC

To resolve the first problem, we decided to use the HDBC package [26]. This package provides some Haskell functions to interact with SQL databases. HDBC has different versions for every type of database. In our system, we choose to use MySQL databases because we use MySQL from the beginning and also because there are no fundamental differences.

As the Listing 4.2 shows us, HDBC provides some data to reach a database. The "MySQLConnectInfo" are all information the package need to connect the program. For example, the host name (mysqlHost) or the name of the database (mysqlDatabase). The HBDC package offers another useful data ("defaultMySQLConnectInfo"). This data defines all parameters of the "MySQLConnectInfo" with the default MySQL settings. For example, the user name will be "root" and the password "".

```
1  data MySQLConnectInfo
2  mysqlHost :: String
3  mysqlUser :: String
4  mysqlPassword :: String
5  mysqlDatabase :: String
6  mysqlPort :: Int
7  mysqlUnixSocket :: String
8  mysqlGroup :: Maybe String
9
10 defaultMySQLConnectInfo :: MySQLConnectInfo
11
12 data Connection
13
14 connectMySQL :: MySQLConnectInfo -> IO Connection
15 connectMySQL defaultMySQLConnectInfo { mysqlHost = "db1" }
```

Listing 4.2: HDBC connection

In our system, we choose to let the user decide. He can define all information in the configuration file to set up the connection. HDBC provides a function and a type for the connection : "connectMySQL" and "Connection". As it is mentioned in the Listing 4.2, the "connectMySQL" function return an monad "IO Connection". Haskell is a functional language, so when you call a function, it calculates the result and returns it. But if, somewhere in your code, you call this function a second time, the previous result will be returned. A function call with the same parameters must return the same result.

This is why, we use this "IO" in the return type [4]. The "IO" monad allows us to represent side-effects, so that the same function can have different results, and also change the context for the posterior function calls. And with databases, we have this problem. When we interact with a database, it is possible that a function with same parameters will return different results. Functions in HDBC are based on that "IO" monad.

When a connection is established with a database, we can retrieve data from it with a SQL request. To do that, the package gives use 4 useful functions : "prepare" , "execute", "executeMany" and "quickQuery". The first one, "prepare", is a function to which you give the SQL request as a string and where you can put some interrogation point (?). The function will return a statement. It is like a model for your request. The second one "execute" will execute the statement with some data. This data will replace the ? . "execute" will return the result of the request. This result is represent by an Haskell data type of HDBC. With the third one "executeMany", the user can give an array of data to execute as many requests as the length of the array. There is a final function which is "quickquery". This one is like a mix between "prepare" and "execute". This function takes a string with a SQL request as a string and returns the result directly.

When a request is executed, HDBC will return a special datatype called "SqlValue". This datatype contains the same type as a MySQL database. For example, a string "mystring" in SQL will be represent "SqlString mystring" in Haskell. The prefix "SqlString" allows us

to identify the type of data, an Haskell constructor-based datatype. And there is many prefixes as there are basic types in MySQL (see Figure 4.3).

```
SqlString String
SqlByteString ByteString
SqlWord32 Word32
SqlWord64 Word64
SqlInt32 Int32
SqlInt64 Int64
SqlInteger Integer
SqlChar Char
SqlBool Bool
SqlDouble Double
SqlRational Rational
SqlLocalDate Day
SqlLocalTimeOfDay TimeOfDay
SqlZonedLocalTimeOfDay TimeOfDay TimeZone
SqlLocalTime LocalTime
SqlZonedTime ZonedTime
SqlUTCTime UTCTime
SqlDiffTime NominalDiffTime
SqlPOSIXTime POSIXTime
SqlEpochTime Integer
SqlTimeDiff Integer
SqlNull

prepare :: IConnection conn => conn -> String -> IO Statement

execute :: Statement -> [SqlValue] -> IO Integer

executeMany :: Statement -> [[SqlValue]] -> IO ()

quickQuery :: IConnection conn => conn -> String -> [SqlValue] -> IO
    [[SqlValue]]
```

Listing 4.3: SqlValue type

4.1.2 Haskell representation

To easily identify and represent changes, we choose to change the format returned by HDBC (SqlValue Listing 4.3). And we decided to re-construct the tables in Haskell. As you can see in the Listing 4.4, we defined the datatype "Table" with a string ("TName"), a list of string ("ColNames") and a two dimension list of SqlValue ("Content"). With this type, we can represent a table like this : Table "name of the table" "name of the columns" "content". Every "Row" needs to have the same length as "ColNames".

```
type TName = String
type ColNames = [String]
type Contents = [Row]
type Row = [SqlValue]

--DB table
data Table = Table TName ColNames Content
deriving(Show, Eq)
deriveBiGULGeneric ''Table
```

Listing 4.4: Database Type

To return this new type, we need to use some small functions to help us (Listing 4.5). "sqlValueToTable" is a function which you can create and return the Table type. This function is used to return a Table type. There is a major function we use in our system in this Listing which is "getTable". This function takes the connection with the database and the name of a Sql Table to return the Table as we define above.

```
1  -- transform an [[SqlValue]]  to a Table
2  sqlValueToTable :: String -> [String] -> [[SqlValue]] -> Table
3  sqlValueToTable name colName xs = if (checkLength colName xs)
4                                    then (Table name colName xs)
5                                    else error("Wrong length of content"
                                        )
6
7  --Get contents of a table with the Table
8  getContentT :: Table -> [[SqlValue]]
9  getContentT (Table _ _ xss) = xss
10
11 --Get columns of a table with the Table
12 getColsT :: Table -> [String]
13 getColsT (Table _ xs _) = xs
14
15 --Get contents of a table
16 getContent :: Connection -> String -> IO[[SqlValue]]
17 getContent conn tname = do
18 quickQuery' conn (selection tname) []
19
20 -- Get columns names
21 getCols :: Connection -> String -> IO [String]
22 getCols conn tname = do
23 des <- (describeTable conn tname)
24 return (map fst des)
25
26 --Return a table
27 getTable :: Connection -> String -> IO Table
28 getTable conn tname = do
29 cols <- getCols conn tname
30 content <- (getContent conn tname)
31 return (sqlValueToTable tname cols content)
32
33 --Return all DB informations
34 getDBTables :: Connection -> IO [Table]
35 getDBTables conn = do
36 tables <- getTables conn
37 sequence (map (getTable conn) tables)
```

Listing 4.5: Small functions to represent data in Haskell

4.2 Propagation of the changes

4.2.1 Synchronization strategy

As said above, this part of the system is one of the part without bidirectional transformations. To keep the consistency, we decided to take only the tables where the user want to share data and check changes. When all changes are identified, we can automatically generate requests to apply these changes in the database (see Figure 4.1).

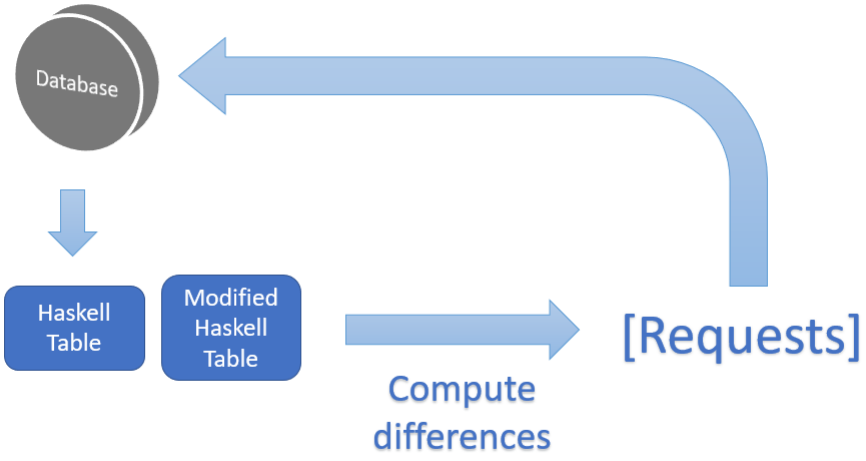

Figure 4.1: Propagation strategy

To generate these requests, we use auxiliary functions (Listing 4.6). First, to easily manage requests, there is a data data type "Request". There are three type of requests (Insert, Drop and Update a row). Also, the request will contain the row where a change is detected. If a new row is created in the modified table, a request "Insert" will be create. The "Drop" request means that the row is missing in the modified table. And the "Update" one means that some "SqlValue" changes in the row.

The first function "createRequest" takes a couple of SqlValue lists and returns the request. The idea is to take the row from the Table and the same row from the modified Table and return a request if needed. For that, we need to be sure of two things, the first one is that the row needs to be the same and also that the "SqlValue" are in the same order. To do that, we use the function "rearrange". This function takes the content of the two Tables and returns a list of couple of rows. This function will put all rows in the same order. Also, to indicate an "Insert" or a "Drop" request, we used the "[SqlNull]" value. The second thing is the order of the columns. This order is managed in the data selection part.

The final function is "separateR" which is the function used to separate the requests. In this function, we want to isolate the contents by the type of request. This function has two parameters. The first one works like a memory to store the result. The second one is the list of requests to separate. The output is also a couple of contents but sorted by request. An "Insert" request will be in first part of the couple and the "Drop" one in the second part. For the "Update" request, we decided to decompose it into two requests (a "Drop" and an "Insert"). We delete the old row in the Table and then, add the updated row in the Table.

```haskell
--DB request type
data Request = Insert Row
        | Drop Row
        | Update Row
        | NoRequest
deriving (Show, Eq)
deriveBiGULGeneric ''Request

createRequest :: (Row,Row) -> Request
createRequest ([SqlNull],ys) = Insert ys
createRequest (xs,[SqlNull]) = Drop xs
createRequest (xs,ys) = if (xs/=ys) then (Update ys) else (NoRequest
    )

find1 :: Row -> Contents -> Row
find1 (x:xs) ((y:ys):yss) = if (x == y)
        then (y:ys)
        else (find1 (x:xs) yss)

rearrange :: Contents -> Contents -> [([SqlValue],[SqlValue])]
rearrange [] [] = []
rearrange [] (y:ys) = ([SqlNull],y):(rearrange [] ys)
rearrange (x:xs) ys = if ((length (filter (== (head x)) (map head ys
    )) ) >= 1)
        then (x,(find1 x ys)):(rearrange xs (delete (find1 x ys) ys)
            )
        else (x,[SqlNull]):(rearrange xs ys)

--separates requests
separateR::(Contents,Contents)->[Request]->(Contents,Contents)
separateR rqs [] = rqs
separateR (is,ds) ((Insert x):xs) = separateR ((x:is),ds) xs
separateR (is,ds) ((Drop x):xs) = separateR (is,(x:ds)) xs
separateR (is,ds) ((Update x):xs) = separateR ((x:is),(x:ds)) xs
```

Listing 4.6: Useful functions to synchronize Table

To simplify, we decide to create a function to regroup every steps. This is the "synchronizeTable" function (Listing 4.7). It will just take a connection to a database and the modified Table. As we use a connection with HDBC, the function needs to return an IO type. With the modified table, we can find the original one simply with the name of the table. After that, we rearrange contents to compare the right rows. With the comparison, Requests are generated. And finally, we execute all the request on the database to propagate all the changes detected.

```
synchronizeTable :: Connection -> Table -> IO ()
synchronizeTable conn (Table tName tCol tContent) = do
tUnmodified <- quickQuery' conn (selection tName) []

rearrTables <- return (rearrange tUnmodified tContent)

view <- return (map createRequest rearrTables)

requests <- return (filter (\x -> x/=NoRequest) view)

sortRequest <- return (separateR ([],[]) requests)
insertRequest <- (insertR conn tName (length tCol))
dropResquest <- (dropR conn tName (head tCol))

executeMany dropResquest (map (\xs -> [head xs]) (snd sortRequest))
executeMany insertRequest (fst sortRequest)
commit conn
disconnect conn
```

Listing 4.7: synchronizeTable

Chapter 5

Data selection and authorization

5.1 Overview

The data being extracted from the database and formatted into a Haskell type, it is now possible for the user to concretely select the data to be shared. He also has the opportunity to set sharing privileges allowing others to edit or not the shared data. For the other members of the sharing group, it is then possible to simply consult the shared data, or for example, to verify the data and (if editable) edit it if necessary. It is in any case important to know that if editable data is shared to a group of people, everyone in this group has to accept the modifications of all the others. This means that at first, the user willing to share data needs to trust updates from everyone in the group. Also, in case of conflicting modifications, the policy of the system is to always chose the last one.

5.2 Description of the selection

This second step is entirely designed to work only with bidirectional transformations coded in BiGUL. The purpose here is to use BiGUL to get a view of the list of *Table* created by the extraction from the database and containing only the selected data for sharing. For every table that the user wants to share, he must give the names of the columns to keep, a selection condition, the name of the column which has to be checked and the boolean telling if he wants the data to be editable or not. This boolean refers to the authorizations granted, namely *"read-only"* or *"writable"*.

5.2.1 Description of the types used

After the data selection, as the content has an additional information to be stored, a new type has been created. Indeed, the authorizations have to be inserted on each row, so that the privileges can be checked when another user is trying to push the update of the row on the network. Thus, a type *TableSync* has been created and is the view of the major BiGUL function described in this section. Listing 5.1 shows the data type as used in the system. The only difference with the *Table* format is on the content of the table, where the list of *Row* is replaced by a list of couple of *Row*, same as before, and *Sync*, the boolean setting the edition privileges on the *Row*. Note that these privileges are set on each *Row* because of the way the data to be shared is stored into a file. When a new sharing process occurs on the same table, the new data is added to the previous one in the file, making the row privileges important.

```
1 type Row = [SqlValue]
2 type Sync = Bool
3 type ContentSync = [(Row,Sync)]
4 data TableSync = TableSync TName ColNames ContentSync
5                 deriving(Show, Eq)
6                 deriveBiGULGeneric ''TableSync
7 data CondType = S String
8                 | I Int
9                 deriving(Show, Eq, Ord)
```

Listing 5.1: The *TableSync* and *CondType* data types

Also note that the Listing 5.1 contains *CondType*, used to represent the visibility condition given by the user. Indeed, *Table* and *TableSync* types both use *SqlValue* instead of standard types. To ease the use of the system, the user is only asked to enter standard types. Currently, the system only deals with *String* and *Int* types. If a developer wants to enlarge the set of types supported, he just needs to add other types to the lines 7 and 8. The syntax of the visibility condition differs slightly from the one described in Chapter 3. Indeed, the evaluation value of the condition is of type *CondType*, which contains a constructor depending on the real type of the value. While the user only gives a *String*, the constructor can be for example *S* or *I*, respectively for *String* and *Int*. Lines 5.1 and 5.2 under display an example of the parameter given to the BiGUL function in charge of the selection.

$$(<= (I\ 1990) \tag{5.1}$$

$$(== (S\ \text{"}Show\text{"})) \tag{5.2}$$

Even though it is mentioned above that the Haskell format of the database may contain many *Table*, the main function used only deals with a single *Table* as source and a single *TableSync* as view. This can be explained by the way the user interacts with the system. Indeed, to ease its use, the system only allows the user to select data from one table at a time, as described in Section 3.2.2, looping when a complete selection has been executed. For each selection made by the user, a complete data transformation is then computed before the user is able to select the data of another *Table*.

It is important to keep in mind that the *TableSync* only contains the columns to keep and the data that satisfies the given condition.

5.2.2 Two-phased implementation

The implementation of the BiGUL function responsible for the data selection has been implemented in two phases. The first one uses many embedded (*emb*) functions and compositions (*Compose*) of these while the second one, respecting the essence of BiGUL, uses *align* functions and only a single *emb* to prepare the *Table* type to be transformed.

Even though the first implementation could concretely operate a selection, it presents two major disadvantages. Firstly, the column on which the selection condition (named visibility condition in the upper sections) is applied must be part of the shared data because of the way in which the BXs have been coded. Secondly, the essence of bidirectional transformations was not really respected, as this first implementation is not a natural way to code in BiGUL. Indeed, the reader could expect the system to use *align* for the selection matter while this first phase uses *emb* functions which are resource consuming due to round-trip verifications. It is then why a second implementation has been coded, deprecating the first one.

5.3 First implementation

This section explains in details the first implementation of the selection. As mentioned above, even though this version could complete required transformation, it is deprecated. This first phase is described, first by explaining the major BiGUL function and then the three smaller composed BiGUL functions.

5.3.1 The major function selecting data

After this global description of what needs to be done and the types used, the main BiGUL function used in this first implementation is presented. As we say "main", it means that auxiliary functions are also used. Indeed, *tableXTableSync* is *composed* of three sub-functions, all coded using BiGUL. But first, it is important to know how to get to each of those sub-methods by explaining in details how this major one works.

```
1  tableXTableSync::  ColNames
2                  -> String
3                  -> (CondType -> Bool)
4                  -> Sync
5                  -> BiGUL Table TableSync
6  tableXTableSync [] _ _ _= error("Nothing to select")
7  tableXTableSync selCols col rowsFct sync = Case[
8  --Error cases
9  $(normal [|\(Table n _ _) (TableSync sn _ _) -> n /= sn|] [p|Table _
       [] []|])
10 ==> error("Table names do not match, please check inputs"),
11 $(normalSV [p|Table _ [] []|] [p|TableSync _ [] []|] [p|Table _ []
       []|])
12 ==> error("Tables have no content, please check input"),
13 --Standard case
14 $(normal [|\(Table n cols _) (TableSync sn _ _)-> (n == sn)&&(
       isSubsequenceOf selCols cols)|] [|const True|])
15 ==> ((selectCont rowsFct col) `Compose` (dropListCols selCols)) `
       Compose` (transformTable sync)
```

Listing 5.2: main method

5.3.1.1 Signature

To use the function, some parameters are required. Note that all those come directly from the user input when he selects what to share, as seen in Section 3.2.1.
First, at line 1, the list of columns to be shared is required. This is the list of columns that should remain in the view after the function is executed in the *GET* direction, from *Table* to *TableSync*. The next parameters are related to the selection condition to know which rows to keep in the view. Indeed, lines 2 and 3 show the need for respectively the name of the column on which the condition is applied and the condition itself. Note that the name of the column is a *string* while the condition is a function. BiGUL being based on Haskell and Haskell being a functional language, it is then possible to use functions as parameter, using the power of the language. The only remaining information needed is the authorization boolean provided by the user, as shown at line 4, *true* meaning *writable* while *false* means *read-only*. Finally, line 5 states that this BiGUL function takes a *Table* as source and a *TableSync* as view. Whatever the direction of the function may be (*PUT* or *GET*), it always returns a BiGUL type being more specifically *Table* or *TableSync*.

5.3.1.2 Error cases

In order to avoid errors when the function is being executed, different cases have been taken into account.

The most obvious one is when there is no column to be selected, i.e. when the view, the *TableSync*, should contain no columns. Such situation would mean that there is no data to be sent or that has been retrieved, which is not logical. Indeed, if there is no retrieved data, it means that there is no current sharing process, because there is no constraint on the data being retrieved from the network. It would then mean that the user possesses the keys representing the data to fetch on the network. Yet, as there is no data to retrieve, the keys do not represent anything. It then raises a logical error. That is why, at line 6 an error message is displayed when the list of columns to select is empty.

The next error cases are no longer on the parameters, but directly on the content of the source and/or the view. That is why they are not written to be pattern matched directly when calling the function. In this part of the system, the name of the *Table* should not change. It then means that a *Table* can be matched to a *TableSync* by its name, as it should be the same and as both types contain this information, as mentioned in Listings 5.1 and 4.4. Indeed, if the names are different, it means that the tables are not the same and thus that the *TableSync* is not the view of the *Table* given as source to the function. As this case should never happen, an error message is displayed as shown in Listing 5.2 at line 9. Note that to check this condition, as both names of the source and view are evaluated together, the *normal* function is used.

The last error case on the other hand uses a *normalSV* function as written at line 11 of Listing 5.2. Indeed, the condition is verified on both source and view, but separately. If one and the other matches, the error is raised. It simply consists in checking if either the columns and content of the source or the view are completely empty. It is however possible to think of the view when the data is deleted from the network, thus making an empty list of content in the view. It may indeed happen. Yet, this is only true for the rows, but not the columns. In fact, a deletion of columns would mean that the structure of the table has been deleted, which is not allowed during a sharing process at this stage in the system. That is why the condition is evaluated on both rows and columns.

5.3.1.3 Standard case

After all those checks and assertions comes the standard case where the selection is concretely going to be computed. Still, to enter this last case, a condition is required. The names of the tables need to match and the columns to select need to be a part of the list of columns of the source, as shown at line 14. The reason of the second condition is obvious, it is not possible to select something that does not exist.
When all those conditions are passed, the system can start computing the composition of three smaller BiGUL functions, each taking care of a major subprocess to entirely select the data. First, the selection of rows is applied, using *selectCont* function at line 15. Then comes the selection of columns, with *dropListCols*. Note that those two functions have a *Table* as source and view. The transformation between *Table* and *TableSync* is only computed in the third function, *transformTableSync*.

5.3.2 Selection of the rows

As previously mentioned, Haskell being a functional language, the condition that every row needs to satisfy to be selected can be a function returning a boolean. This parameter is then given from the major function to this one, as it is the case for the name of the column on which the condition is applied, as shown in Listing 5.3 respectively at lines 4 and 5. There is however a constraint. This method being bidirectional, it implies that both the source and view should contain the column on which the condition is applied. Otherwise, inconsistency occurs, as the round-trip property and the *GETPUT* and *PUTGET* properties are not verified anymore. As mentioned in Section 5.2.2, this is a critical limitation partially responsible for the second implementation as *align* functions do not require such constraint.

As it is possible to see in Listing 5.3 at line 7, *selectCont* is coded using the *emb* function. Why is it the case? First, notice the type of the parameters. The column on which the condition is applied is a string. Yet, the content of a table is a list of *SqlValue* where only an index can mean something. In this case, there is two options. Use intermediate BiGUL functions to transform the content to set the name of the column on every value of each row, or to use *emb* to be able to call external Haskell function converting the string in integer. As *emb* is easier to code and understand in this case, this option has been implemented. It is then possible to see that the *GET* and *PUT* direction have been coded, respectively at lines 9 and 12. Both directions call a sub-function doing the concrete job, respectively using the keywords *get* and *put* to execute it. In each direction again, a case is made on the result returned by the function. Indeed, if it returns *Nothing*, it means that an error occurred and is displayed even though it should not happen. Otherwise, the content is returned and then used to construct a *Table*. Note that this sub-function described in Listing 5.4 only works on the list of rows, taking the index and the condition as parameters, while *selectCont* is used to construct *Table* and retrieve the needed index.

```
1  {-
2  Function to select the content (i.e. select the rows) given Table(s)
       . An emb function is used in order to retrieve indexes and be
       able to use the columns in the update function for the content.
3  -}
4  selectCont :: (CondType -> Bool)
5          -> String
6          -> BiGUL Table Table
7  selectCont rowsFct colSel = emb g p
8          where
9                  g (Table n cols conts) = Table n cols (case(get (
                       selectContent (getIndex colSel cols) rowsFct)
                       conts)of
10                       Nothing -> error("An error occurred during
                             the selection of the rows GET")
11                       Just content -> content )
12                 p (Table _ _ sconts) (Table n cols conts) = Table n
                       cols (case (put (selectContent (getIndex colSel
                       cols) rowsFct) sconts conts) of
13                       Nothing -> error("An error occurred during
                             the selection of the rows PUT")
14                       Just content -> content )
```

Listing 5.3: row selection major function

This sub-function works with lists of rows, either as source and view, as previously mentioned. In order to match a row with another, the lists need to be ordered in the same way, even if they do not contain the same data. Yet, as a DBMS is used, both lists should always be ordered by the same criteria. Following this logic, if two rows from the source have not been updated in the view, they always should be in the same order in the source and in the view.

As the parent function has extracted the index, *selectContent* has all the needed information and is therefore not obligated to use a *emb* function. It is then implemented with *Case*, where the first condition is the empty list scenario where nothing should be updated. In order to really understand the next conditions, it is critical to remember that BiGUL functions are coded in the *PUT* direction. Also, even though each row contains a primary key, it is possible that the selection condition is not applied on it but on other columns. Following this principle, it is possible that rows matching on primary keys in this BiGUL function do not satisfy the selection condition anymore, because the columns in the view may have been updated. Indeed, only the primary keys are never editable. To change the primary key, a new row with a new primary key needs to be created and the rest of the initial row can be copied.

```
1  selectContent ::  Int --The index of the column to satisfy condition
2          ->  (CondType -> Bool)
3          ->  BiGUL Content Content
4  selectContent index rowsFct = Case[
5  $(normalSV [p|[]|] [p|[]|] [p|[]|])
6          ==> $(update [p|[]|] [p|[]|] [d||]),
7  -- the column on which the condition is applied is true in both
       source and view.
8  $(normalSV [|\(row:rows) -> rowsFct $ setCondType (getElemAtIndex
       index row)|] [|\(row:rows) -> rowsFct $ setCondType (
       getElemAtIndex index row)|] [|const True|])
9          ==> $(update [p|row:rows|] [p|row:rows|] [d|row = Replace ;
              rows = selectContent index rowsFct|]),
10 -- the condition is not true on the source and that source is not in
        the view.
11 $(normal [|\(row:rows) rowVs -> (not (rowsFct $ setCondType (
       getElemAtIndex index row))) && (notElem row rowVs)|] [|const True
       |])
12         ==> $(update [p|_:rows|] [p|rows|] [d|rows = selectContent
              index rowsFct|]),
13 -- the condition is not true on the source, but the row is in the
       view (has newly been inserted in the source, see adaptive).
14 $(normal [|\(row:rows) rowVs -> (not (rowsFct $ setCondType (
       getElemAtIndex index row))) && (elem row rowVs)|] [|const True|])
15         ==> $(update [p|row:rows|] [p|row:rows|] [d|row = Replace ;
              rows = selectContent index rowsFct|]),
16 -- if all conditions above fail, and the view is not in the source,
       then source gets it.
17 $(adaptive [|\rowSs (rowV:rowVs) -> notElem rowV rowSs |])
18         ==> \rowSs (rowV:_) -> rowV:rowSs
19 ]
```

Listing 5.4: row selection sub-function

Knowing all that, the second case at line 8 of Listing 5.4 is applied when the selection condition is true on rows from both source and view. In this case, the content is simply updated. The third case at line 11 is applied when the row in the source does not satisfy the selection condition and is not in the view. The row is simply not taken into account and skipped. If those conditions are not matched, an adaptive case is executed. However, it has a condition too. Indeed, the current row of the view (the first one thus) must not be in the source, i.e. in the list of rows taken as source. Such condition is evaluated thanks to the primary keys, as previously mentioned. If it is proven true, the source is simply added this new row as the head of the list. *selectContent* is then called again with the rearranged source and the view. Yet, no case previously explained is going to fit the case here, because the first row of both source and view is the same and most likely do not match the selection condition. This situation is checked at line 14. In other words, when the data is already shared and must be put back in the database (in other words, pulled from the network, the *PUT* way), it is also possible that the view contains a row which is not in the source. In that case, the source is just updated by adding the missing row.

5.3.3 Selection of the columns

5.3.3.1 Overview

In the *GET* direction, after the row selection comes the selection of columns following the list given by the user. Concretely, selecting a set of columns of a table is the same as dropping all columns that are not in the wanted set. As such method was already partially implemented during the preliminaries of the system, it has been reused here. As the user gives a list of column names as parameters, the BiGUL function has then to drop all the columns that are not in the list. Yet, in order to keep the data identifiable, the user needs to keep the primary key, set by default to be the first column. Otherwise, it would be possible that after the application of this function, two rows have exactly the same content, which raises a problem in case of update and reconciliation.

5.3.3.2 Sub-functions description

Once again, as we needed to work with indexes, *emb* functions are used to be able to use Haskell methods. Yet, as there are lots of cases to deal with to make the function as general as possible, the complexity is quite high. To better understand, it is important to know what those Haskell functions do, even though it may be deducted by their name. Below can be found a brief explanation of those methods. The code can be found in Annex .1.

- *removeListCol* : removes an element from a list based on its index. The index is then needed and a counter needs to be initialized.
- *removeListContent* : removes an element from a double list, still based on the index of the deepest element. For each element of the first list (each one being a list, then), *removeListCol* is called, and the index given as parameter.
- *getIndexesList* : verifies the presence of an element in both lists given as parameters. If the element of the first list is in the second one, its index in the second list is added to the list of indexes returned.
- *elemsNotInBoth* : takes two lists as parameters, returns a list containing the elements of the second list (as parameter) not being in the first one. More concretely, in this case, this method is used to get the list of the columns that must be dropped. Indeed, as "*Colname*" contains the columns to keep, an "anti-list" must be constructed.

5.3.3.3 Concrete implementation of the columns selection

In the *GET* direction written at line 9 of Listing 5.5, only the functions described above are used.
To create the list of columns of the view, the procedure is the following. First, the list of columns to drop is created using *elemsNotInBoth*. Then, this list is used to get the indexes of the columns to be dropped in the list of columns of the *Table* as source, using *getIndexesList*. Finally, having the indexes, the columns can be removed using *removeListCol*.
The procedure is exactly the same to create the list of rows of the view, but *removeListCol* is replaced by *removeListContent*, as the content is a list of *Row*.

```
1  {-
2  This function will delete from the Table all the data at the indexes
       given in parameter. ColNames parameter is the list of columns to
       KEEP
3  -}
4
5  dropListCols :: ColNames -- The list of columns to KEEP
6  -> BiGUL Table Table
7  dropListCols colName = emb g p
8  where
9  g (Table n cols conts) = Table n (removeListCols (getIndexesList (
       elemsNotInBoth colName cols) cols) 0 cols) (removeListContent (
       getIndexesList (elemsNotInBoth colName cols) cols) conts)
10 p (Table n cols conts) v = insertColumns (Table n cols conts) v (
       getIndexesList colName cols) 0
```

Listing 5.5: columns selection functions

The *PUT* direction is coded using a single major function named *insertColumns* shown at line 10. It is critical to notice that here the function *elemsNotInBoth* is not used, meaning that the list of indexes produced contains the indexes of the columns to keep. It is then very different from the *GET* direction where the indexes of the columns to drop are used.

When this function is executed in the *PUT* direction, opposite than the one described above, the goal is to reconcile the view (the selected *Table*) with the source (the *Table* containing all columns) to produce an updated source. As seen in Listing 5.5, the major function used to create the updated source is called *insertColumns*, as shown in Listing 5.6, at line 4. Such function is indeed interesting for the pattern matching provided.

```
1  {-
2  This method is used to concretely implement the PUT direction of the
       dropListCols function. Created to use the pattern matching.
       IMPORTANT : in this case, the [Int] is the list indexes TO KEEP
3  -}
4  insertColumns :: Table -> Table -> [Int] -> Int -> Table
5  insertColumns (Table _ [] _) (Table n cols conts) _  _ = Table n
       cols conts
6  insertColumns (Table n cols conts) (Table _ [] _) _ _ = Table n cols
       conts
7  insertColumns (Table _ _ _) (Table n cols conts) [] _ = Table n cols
       conts
8  insertColumns (Table _ cols conts) (Table n scols sconts) idxs count
       = Table n (insertCol idxs 0 cols scols) (insertColDbl idxs conts
       sconts)
```

Listing 5.6: columns selection functions

Listing 5.6 shows that the list of columns and the list of *Row* of the *Table* are updated using different methods. The basic structure of those elements are indeed different.

The updated source must contains the columns of the view augmented by the complementary columns of the original source. To do so, the major function calls *insertCol*, as shown in Listing 5.7 under. The goal here is to create an updated source by placing the columns of the original source at the right place in the view. Obviously, if a column is present in both source and view, only the view is kept, as line 15, where *scol* is the current column of the view. The algorithm used is the following. The list of columns of the view is browsed and a counter is increased for each element seen. The main condition is on the counter matching integers in the list of indexes. When the value of the counter is in the list, a the column at this precise index in the source must be included in the view. Also note that all different cases met on columns lists are taken into account, using pattern matching again.

```
1  {-
2  This method takes two lists as parameter, list 1 and 2, where 2
       should be shorter or equal to list 1 and should only contain
       elements of 1. The [Int] is the list of indexes, Int is the
       current index.
3  -}
4  insertCol::(Eq b)=>[Int] -> Int -> [b] -> [b] -> [b]
5  insertCol [] _ _ scols = scols
6  insertCol _ _ [] [] = []
7  insertCol idxs i cols [] = if(i == 0) then cols
8          else
9                  if(i< length cols) then
10                         (getElemAtIndex i cols):(insertCol idxs (i
                               +1) cols [])
11                 else []
12 insertCol idxs i cols (scol:scols) =
13         if(i < length cols)then
14                 if(elem i idxs)then
15                         scol:(insertCol idxs (i+1) cols scols)
16                 else (getElemAtIndex i cols):(insertCol idxs (i+1)
                       cols (scol:scols))
17         else []
```

Listing 5.7: columns selection functions

The updated source must also contain the rows of the view augmented by the complementary rows of the original source. To do so, *insertColumns* calls *insertColDbl*, "Dbl" standing for "double", such as the list of rows, as shown in Listing 5.8. Compared to the update of the list of columns, updating the content is more tricky. Indeed, the list to work on is a list of lists. Moreover, the data perhaps being editable, it is possible that a row which is not in the source has been added to the view. Yet, as the view does not contain as much information as the source does, the row cannot just be inserted into the new source. Indeed, no data is set for all the columns previously dropped (in the *GET* direction). The data should then be either set to a default value, filled by the user or matched to some existing row that has not been shared. The system uses the first option, namely setting the unknown columns to a default value defined in the configuration file. It is obvious that the default values must match the type of the columns they are put in. Another option would have been to ask the user to fill those blanks himself, which in the case of big databases is not really efficient.

Following those restrictions, *insertColDbl* needs to call the *giveRightRow* function, as shown at line 16 of Listing 5.8. This method actually checks if a row of the view has a match in the source by trying to match both primary keys. The *filter* function is used to get the intersection of the primary keys of the row as view and the source.

On one hand, if the intersecting set is not empty, the matching row is returned, allowing *insertColDbl* to continue its execution by calling *insertCol*. Indeed, this last function has been coded using generic types, making it easy to be reused. It is then possible to understand the comment at line 7 in Listing 5.7. In the case there is no row at all in the view, then the shared columns would be deleted from every sharing member's database, even the owner of the data. As this deletion process should not be allowed without the agreement of all the members of the sharing process, or at least, of the owner of the data, deletion of rows is simply forbidden. It may yet raise other ethic, legal or sharing concerns. To forbid such process, line 7 states that when the list of the view is empty, then the list of the source is returned.

If the intersecting set is empty, default values need to be included at the right place in the row of the view. To do so, *createDefaultRow* is called. Using the list of indexes and a counter, just as *insertCol*, the default values are inserted at the right place. Obviously, types are matched, using the *defaultSqlValue* method which can be found in Annex .1.

```
1  {-
2  This method inserts columns in rows, double lists of SqlValue. For
       each element of the list as second parameter (view in this case)
       the columns are inserted at the right index (each element being a
        Row (list of SqlValue)).
3  -}
4  insertColDbl ::(Eq SqlValue) => [Int] -> [[SqlValue]] -> [[SqlValue
       ]] -> [[SqlValue]]
5  insertColDbl [] _ sconts = sconts
6  insertColDbl _ _ [] = []
7  insertColDbl _ [] sconts = sconts
8  insertColDbl idxs conts (scont:sconts) = (insertCol idxs 0 (
       giveRightRow conts scont idxs) scont):(insertColDbl idxs conts
       sconts)
9
10
11
```

```
{-
This method finds, in a list of rows, the element (row) matching the
    row given in parameter using the identifier of each row. If none
    is found, a default row is created.
Preconditions : the identifier of the view (unique row parameter) is
    the head element of the row, and the identifier's index of the
   source (list of rows) is at the head of the list of indexes to
   KEEP.
-}
giveRightRow:: (Eq SqlValue) => [[SqlValue]] -> [SqlValue] -> [Int]
   -> [SqlValue]
giveRightRow _ [] _ = []
giveRightRow [] scont _ = scont
giveRightRow rows scont idxs =
if((filter (\x -> (head scont) == (getElemAtIndex (head idxs) x))
   rows) == []) then
--return scont modified to have the right length
createDefaultRow (head rows) scont idxs 0
else
head $ filter (\x -> (head scont) == (getElemAtIndex (head idxs) x))
    rows

{-
This method creates a row having default values in columns not
   present in the view. A standard row is taken from the source to
   determine the type of the columns where the unknown values should
    be inserted.
-}
createDefaultRow ::(Eq SqlValue) => [SqlValue] -> [SqlValue] -> [Int
   ] -> Int -> [SqlValue]
createDefaultRow [] [] _ _ = []
createDefaultRow row [] idxs i = if(i==0) then []
        else
                if (i < length row)then
                        (defaultSqlValue (getElemAtIndex i row)):(
                           createDefaultRow row [] idxs (i+1))
                else []
createDefaultRow row (scont:sconts) idxs i = if(elem i idxs)then
                scont:(createDefaultRow row sconts idxs (i+1))
        else (defaultSqlValue (getElemAtIndex i row)):(
           createDefaultRow row (scont:sconts) idxs (i+1))
```

Listing 5.8: columns selection functions

5.3.4 Authorizations in the selected data

The last part is the real transformation from the completely selected *Table* to the *TableSync* including the boolean controlling the authorizations. This third function called by *tableXTableSync* shown in Listing 5.2 simply converts the source type into the view type and vice-versa, namely *Table* and *TableSync*. In other words, this BiGUL function simply adds to every row of the *Table* the authorization boolean given by the user. As reminder, a *false* value indicates a read-only content while a *true* one indicates that the content can be edited by everyone.

transformTable is implemented using an *emb* function, as described in Listing 5.9, for its easiness with small functions. Indeed, if a *Case* were used, two BiGUL method would be required, as the first one would deal with the *Table* type, and the other one with the rows

themselves. It would then highly increase the complexity and the "only code *PUT*" property would not be an advantage anymore. That's why *emb* is used, combined with a standard Haskell *map* function on the rows to add the boolean on each row, as shown at lines 7 and 8.

```
1 {-
2 Method that changes the type from Table to TableSync in the GET
     direction and from TableSync to Table in the PUT direction.
3 -}
4 transformTable :: Sync -> BiGUL Table TableSync
5 transformTable sync = emb g p
6         where
7                 g (Table n cols rows) = TableSync n cols (map (\row
                      -> (row,sync)) rows)
8                 p s (TableSync sn scols srows) = Table sn scols (map
                      (\(row,sync) -> row) srows)
```

Listing 5.9: function converting *Table* to *TableSync* and vice-versa

It is important to know that even though bidirectional transformation is here used to consistently convert the *Table* into a *TableSync*, the editing authorizations are not verified in this part of the system but in the translation.

5.4 Second implementation

Despite all justifications, the first implementation has been deprecated for its constraints and for its code. We then came with a second one, more in the essence of BiGUL. This second version is coded using *align* methods introduced in Chapter 1. Another advantage of this solution is that it is shorter, more readable and more understandable.

As it is possible to see thanks to the Listings of this section, there are a few unidirectional functions coded. Those are meant to prepare the source data structure (*Table*)to be *align*ed with the one of the view (*TableSync*). The information whether a row or a column is selected to be shared is boolean. Indeed, it is either selected or not, *True* or *False*. A boolean has then been chosen to differentiate the shared content from the unshared one. The structure of the data of the source has then to be adapted :

$$Content \quad => \quad [\ (Bool\ ,\ [(Bool, SqlValue)])\ ] \tag{5.3}$$

In the new structure (on the right in Formula 5.3), the first boolean is used to check whether the Row itself is selected. A *True* value is used in that case. The second boolean is set on every value of a Row, where *True* means that the column is selected to be shared.

The code of the selection is going to be explained, first in the *GET* direction, as it uses all the "structure changing" functions of this part of the system. The main function is going to be explained, followed by the functions it calls, leading in the end to the completion of the selection of the data to be shared. Note that before explaining the *align* functions, the *PUT* direction of the main function is explained. It allows us to describe these *align* functions in this direction, as it is may be more understandable to the reader.

5.4.1 The main function selecting data

This major function consists in transforming the source structure using several unidirectional functions to obtain the structure of Formula 5.3. To concretely do this transformation and call those functions, this major selection is coded using an embedded function, allowing, despite higher resource cost, to set the values of the structures differently between *GET* and *PUT*.

As it is possible to see in Listing 5.10, the parameters of the major *emb* function are the same as in the previous implementation phase, detailed in Section 5.2. Note that Listing 5.10 deliberately does not show the *PUT* direction of the *emb* function as it is described further.
If the list of columns to share is empty, an error is thrown, as shown at line 6. Otherwise, the embedded function starts. The *GET* direction then takes a *Table* as source and needs to return a *TableSync* as view. As the name of those tables does not change, the name of the source is set to the view. The list of columns of the *TableSync* is simply set to be the parameter given to the function: the list of columns to share. The *Content* needs now to be selected and changed to the *ContentSync* format, as shown at line 9. But first, the structure of the source needs to be changed in order to be able to efficiently use an *align* function.
The next section explains then the structure reformatting and will come back to this major function.

```
1  selectionT :: ColNames
2             -> String
3             -> (CondType -> Bool)
4             -> Sync
5             -> BiGUL Table TableSync
6  selectionT [] _ _ _= error("Nothing to select")
7  selectionT selCols colSel fct writable = emb g p
8    where
9          g (Table tname cols cont) = TableSync tname selCols (map (\
               row -> (row,writable)) (fromJust (get (selectionCont
               selCols cols)  newCont)))
10           where
11             newCont = (makeNewCont (findIds selCols cols 0) (findIds
                   [colSel] cols 0) fct cont)
```

Listing 5.10: The main BiGUL function selecting the data in the *GET* direction

5.4.2 Reformatting the structure of the source

This section explains how the structure has been changed to match Formula 5.3. In Listing 5.10, at line 9, the function *newCont* defined at line 11 is referred to. This last line shows that *newCont* calls *makeNewCont* in order to change the *Content* of the source.

This latter unidirectional function shown at line 7 of Listing 5.11 takes 4 parameters.
The first one, a list of *Int* is the list of indexes of all the selected columns in the complete table, calculated using the *findIds* function at line 1. This latter one creates a list with the indexes of the columns being both in the selection list and the complete list of columns of the source. Note that the list consists in the indexes of the columns when they are inside the complete list.

The second parameter is the index of the column on which the selection condition is evaluated. It is set as a list in order to use the function *findIds*. The third and fourth ones are respectively the selection function (the condition) and the *Content* of the source.

For each row of the *Content*, the function checks whether the row must be selected by applying the selection function to the value at the index given by the second parameter, value transformed from *SqlValue* to *CondType*. The boolean returned by this selection function is then directly set and is the boolean attached to the *Row*. To attach a boolean to each column of the row, another function, *colSelect* is called.

Given the list of indexes of the columns to share (the first parameter of *makeNewCont*), the *Row* of the source and a counter, *colSelect*, shown at line 10, is able to add the boolean on each column of the *Row* given in parameter. Each column of the *Row* is checked to be in the list of indexes or not. If the column is not shared, i.e. it is not in the list of indexes, the boolean of the column is set to *False*. It is of course set to *True* otherwise, thanks to the *elem* function provided by Haskell.

Thanks to those 3 functions, the *Content* of the source is transformed into the new structure and is ready to be concretely selected using *align* functions.

```
1  findIds :: [String] -> [String] -> Int -> [Int]
2  findIds _ [] _ = []
3  findIds selCols (c:cols) id = if (c `elem` selCols)
4          then id:(findIds selCols cols (id + 1))
5          else findIds selCols cols (id + 1)
6
7  makeNewCont :: [Int] -> [Int] -> (CondType -> Bool)  -> Content ->
       [(Bool,[(Bool,SqlValue)])]
8  makeNewCont ids (id:[]) fct cont = map (\row -> ( fct (setCondType (
       row !! id)),(colSelect ids row 0))) cont
9
10 colSelect :: [Int] -> Row -> Int -> [(Bool,SqlValue)]
11 colSelect _ [] _  = []
12 colSelect ids (r:row) i = ((i `elem` ids),r):(colSelect ids row (i
       +1))
```

Listing 5.11: The functions to reformat the structure of the Content of the source

5.4.3 Back on the main function

Before explaining the *align* functions, the *PUT* direction of the main function is firstly described in Listing 5.12. It will then be possible to fully understand *align* functions.

As in the *PUT* direction, the goal is to transform a *TableSync* to a *Table* by merging both content and updating the one of the *Table* if necessary, the structure of the *Table* is kept. Indeed, the name and list of columns returned is directly taken from the source parameter, as shows line 9. The content has yet to be merged and updated. To do so thanks to the *align* functions, the structure of the *Content* of the source has, again, to be modified following Formula 5.3. Yet, as the source must contain all the *Row*s, the first boolean of the new structure is always set to *True*, as shown at line 12 of Listing 5.12. Using the same *colSelect* function previously described, the booleans are set on each column of each *Row*. Note that the third *map* function at line 10 is used because of the *ContentSync* induced by the *TableSync*. The *Content* then needs to be extracted in order to be given as parameter

to the *align* function.

After the call of these *align* functions by executing the *selectionCont* function at line 10, the *Content* of the new source needs to be extracted. Indeed, this latter function returns the new structure of the content of the source because of the *PUT* direction. That is the purpose of the first and second *map* functions at the same line previously mentioned.

```
1  selectionT :: ColNames
2             -> String
3             -> (CondType -> Bool)
4             -> Sync
5             -> BiGUL Table TableSync
6  selectionT [] _ _ _= error("Nothing to select")
7  selectionT selCols colSel fct writable = emb g p
8    where
9          p (Table nS cS contS)(TableSync nV cV contV) = Table nS cS
10         (map (\x -> map (snd) (snd x)) (fromJust (put (selectionCont
               cV cS) newContS (map fst contV))))
11           where
12             newContS = (map (\c -> (True,(colSelect (findIds selCols
                   cS 0) c 0))) contS)
```

Listing 5.12: The main BiGUL function selecting the data in the *PUT* direction

5.4.4 Selection with alignment

In either *GET* or *PUT* direction, when the structure transformations have been completed, the *selectionCont* function is called.
This function, shown at line 2 of Listing 5.13, uses the *align* function explained in Chapter 1. It is used to align the rows and update them.
The selection condition for a *Row* in the new structure is to have the boolean set to *True*. This is mostly important in the *GET* direction as in the *PUT* one, all booleans concerning the row are set to *True*. Then, a source and a view are matched thanks to the primary key, which must be the first column of the row.

This *selectionCont* function takes 2 parameters, namely the list of columns to share and the whole list of columns. They are important for the reconstruction of a row when a view contains a new row which is not in the source. In that case, there is indeed no information about the missing columns in the view. So, *makeNewCol* shown at line 33, inserts, thanks to those two parameters, a couple of a boolean (*False*) and *SqlNull* when it is needed. When a column is in both source and view, the boolean is set to *True* and the value of the column in the view replaces the *SqlNull* value.
Note that instead of inserting *Null* values is to recreate a row in the source from the information available in the view, it was possible to use the function described in the first phase and use default values.

Another function needed by *align* is about the deletion of an item from the source. In this case, as booleans are used, to not consider the row in the *align* function, the boolean of the row is set to *False*. The row itself is then never really deleted from the source.

As each row is composed of different columns which can be selected or not, the update function calls another function, *selectionContCols*, also using *align*, as shown from line 18. This latter function is very similar to the previous one. Indeed, instead

of dealing with a list of rows in the new structure and the *Content* type, this function aligns a row in the new structure (a list of couples of a boolean and a *SqlValue*) with a *Row*.

```
--align the content
selectionCont :: [String] -> [String] -> BiGUL [(Bool,[(Bool,
    SqlValue)])] Content
selectionCont selCols cols = align
        -- Select only rows where the first Bool is True
        (\(b,_) -> b)
        --match between the source and the view (by PK, at first pos
            )
        (\(_,rowS) rowV -> ((snd (head rowS)) == (head (rowV))) )
        --update function
        $(update [p| (_,row) |]
                 [p| row |]
                 [d| row=selectionContCol |])
        --create a source from a view (need to recreate a COMPLETE
            row)
        (\row -> (True,(makeNewCol selCols cols row)))
        --delete an item from the source list (make it unshared, no
            real deletion)
        (\(b,row) -> Just (False,row))

--align the row in the content
selectionContCol :: BiGUL [(Bool,SqlValue)] Row
selectionContCol =  align
        -- select only columns where Bool is True
        (\(b,_) -> b)
        --match between the source and the view
        (\(b,sqlValS) sqlValV -> b )
        --update function
        $(update [p| (_,sqlVal) |]
                 [p| sqlVal |]
                 [d| sqlVal=Replace |])
        --create a source from a view
        (\sqlVal -> (True,sqlVal))
        --delete an item from the source list (make it unshared, no
            real deletion)
        (\(b,sqlVal) -> Just (False,sqlVal))

makeNewCol :: [String] -> [String] -> Row -> [(Bool,SqlValue)]
makeNewCol _ cols [] = (map (\_ -> (False,SqlNull)) cols)
makeNewCol _ [] (_:_) = error("Column retrieve missing")
makeNewCol selCols (c:cols) (sqlV:xs) = if (c `elem` selCols)
        then (True,sqlV):(makeNewCol selCols cols xs)
        else (False,SqlNull):(makeNewCol selCols cols (sqlV:xs))
```

Listing 5.13: The *align* functions used to concretely select the data to share

Still mostly important for the *GET* direction, the column of the source is only selected to be in the view when the boolean is *True*.
A match between the source and the view can only be obtained by the boolean set on each value of a row. Indeed, it is possible that the value itself of the source and view will never match for the data of a row (except for the primary key). In that case, the only way to know if a value was originally both in source and view is to use the boolean, where *True* means that it was originally in both source and view.

The update function simply consists in a *Replace* function, also described in Chapter 1.
In order to create a source from a view, it is needed to add *True* to the value (making a couple of these two) in order to match the format of the source.
When an item needs to be deleted from the source, like in *selectionCont*, the boolean is just set to *False*, not really deleting any data from the source.

Chapter 6

Universal Format

6.1 Overview

Now with the *TableSync*, we want to find a way to share the information. The first idea was to share data between two different databases. To do that, we thought to share the *TableSync* and a mapping table. This is the shared format strategy. After this, we realized that our method could be extended to multiple databases. This is the universal strategy. This new method forced us to change the way of sharing. The mapping table couldn't be shared but stored locally for each peer.

6.2 Shared Format Strategy

The first idea was the shared format strategy. This idea consists on sharing the *TableSync* and a mapping table. The mapping table is a table to match the ID of tables, just like the Hyperion project [12]. With the mapping, we create a shared format of the table: a table understandable by the two peers. As we want to keep the consistency and also share/retrieve data, the bidirectional transformations seems useful. The first transformation will add information if it is possible. And the second one will check the information in the mapping table and add it in the databases. To be clear, we want to create a shared format table with the mapping table and use this new table to add information in databases. This shared format table will stay in the network and used only for sharing more data or retrieving the table.

6.2.1 Shared Mapping

As I mentioned previously, this method is based on the Hyperion project. As you can see in the figure 6.1, we have two databases (AA and BB).

AA_Passenger

pid	name
1	Renee
2	Verena
3	Iluju

AA_Ticket

pid	fno	meal
1	AA210	Meat
2	AA378	Veg.

AA_Flight

fno	date	dest	sold	cap
AA210	01/05/03	L.A.	120	256
AA341	01/15/03	N.Y.	160	160
AA378	01/21/03	S.F.	90	124

BA_Passenger

pid	name
1	John
2	Renee

BA_Fleet

aid	type	capacity
B-1	Boeing 747	340
B-2	Boeing 737	130
B-3	Boeing 737	107

BA_Flight

fno	date	to	sold	aid
BA1023	01/07/03	LAX	67	B-3
BA1078	01/15/03	JFK	118	B-2
BA1109	01/15/03	ORD	164	B-1

BA_Reserve

pid	fno
1	BA1023
2	BA1078
2	BA1109

Figure 6.1: Hyperion Project data

fno_{AA}	fno_{BA}
AA210	BA1023
AA341	BA1078
AA341	BA1080

Figure 6.2: Hyperion : mapping table

With this data, the project wants to share flight information. For example, we can imagine that AA company bought BA company. The two companies stay independent but AA wants to know BA_Reserve of the BA company. With that information a flight can be available in the two companies and the passengers can reserve in the two companies for a same flight. In the Hyperion project, they used a mapping table (figure 6.2). This table is used to make a link between two tables in two different databases. With the *fno* (the primary key of the table) of the AA_Flight, we know the *fno* of the BA_Flight. We decided to use the same idea. Find the primary key of the table and associates it with the primary key of another table. This association is stored in the mapping table just like the Hyperion project.

6.2.2 Local Format to Shared

First, we decided to simply replace the primary key of the first table with the primary key of the second one. In that case, the mapping needs to be the same for the two databases. This method is very complex because you need to specify a major database. Indeed, with bidirectional transformations, we have the get and the put direction. With a same mapping table, the BiGUL function will return always nothing for one peer. The problem is that we stock the same mapping table as a SQL table in databases. It is impossible to create a BiGUL function which 2 different *GET* and 2 *PUT*. As we want to have the same code for any peer, the *PUT* function will be format A to format B for the peer A and format B to format A for peer B. This idea was not the good one.

After that we chose to create a shared format. We use a common format between the two peers to keep a bidirectional transformation and then the consistency. The idea is represented in the figure 6.3. With the mapping table, we can create new primary keys. We decided also to add the column name and the table name in the mapping table.

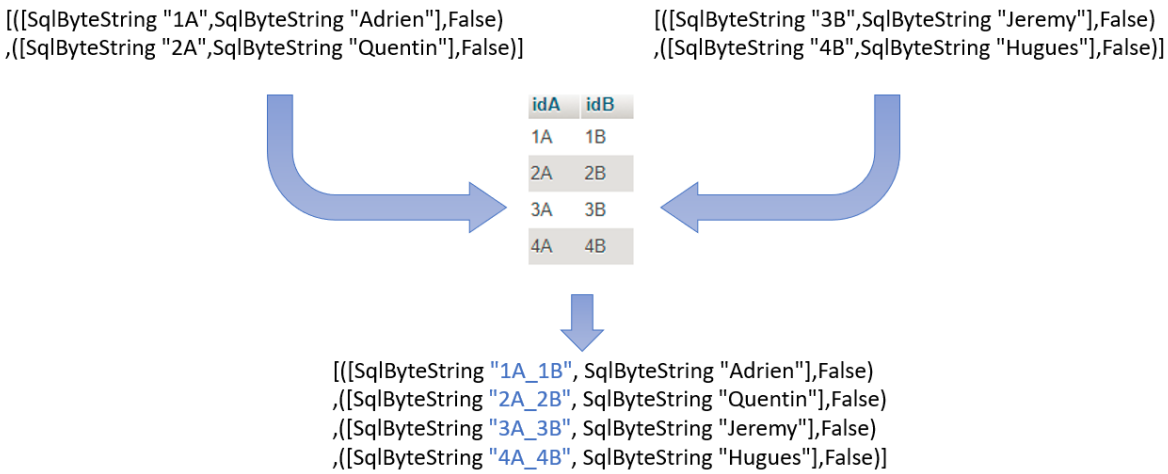

Figure 6.3: Local Format to Shared

In this situation, the *PUT* function (to translate into shared format) is consistent. With a same mapping table and the BiGUL, we are sure that the consistency is preserved. This BiGUL function is *translateTableSync* in the Listing 6.1.

```
--Translate a TableSync list into a TableSync list with an shared format
translateTableSync :: Int -> [[SqlValue]] -> BiGUL [TableSync] [TableSync]
translateTableSync i (id:ls) =
Case [
$(normalSV [p| [] |] [p| [] |] [p| [] |])
        ==> $(update [p| [] |] [p| [] |] [d| |])
,$(normalSV [|(\((TableSync n _ _):_) -> (n==(fromSql(joinSql id))))|]
        [|(\((TableSync n _ _):_) -> (n==(fromSql(head(cond i id))))|]
        [|(\((TableSync n _ _):_) -> (n==(fromSql(joinSql id))))|])
        ==> $(update [p| (TableSync n cols cont):xs |]
                [p| (TableSync n cols cont):xs |]
                [d| n=(joinName i id); cols=(joinCols i ls) ;
                cont=(translate i ls) ;
                xs=(translateTableSync i ls) |])
,$(normalSV [|(\((TableSync n _ _):_) -> (n/=(fromSql(joinSql id))))|]
        [|(\((TableSync n _ _):_) -> (n/=(fromSql(head(cond i id))))|]
        [|(\((TableSync n _ _):_) -> (n/=(fromSql(joinSql id))))|])
        ==> translateTableSync i (lastP id ls)
,$(adaptiveSV [p| _:_ |] [p| [] |])
        ==> (\s _ -> [])
,$(adaptiveSV [p| [] |] [p| _:_ |])
        ==> (\_ ((TableSync n _ _):_)
        -> (TableSync (fromSql(idSync (toSql n) (id:ls))) [] []):[])
]
```

Listing 6.1: Translation Local and Shared formats

translateTableSync is coded with three *normalSV* and two *adaptiveSV*. As almost all *Case*, the first *normalSV* is the base case. The others are just like a *if*. If the primary key is the same as in the mapping table, we join the table name, columns names and the primary key to create a *TableSync*, and continue to translate the others *TableSync*. If not, we progress in the mapping table. The two *adaptiveSV* are also classic. The first one is here to delete all source elements that are not in the view. The other will create a new TableSync for the source. As it is a bidirectional transformation, we can use the *PUT* to create our shared format and also the *GET* to recover the local format.

6.2.3 Shared Format to Local

As we coded the *translateTableSync* function, the *GET* is also here to check if there is no mistake during the translation. For example, in the figure 6.4, if we have a wrong primary key in the mapping table or in the shared format, the row will be rejected by the function and return nothing for the row. In this case, the *GET* part is like a checker for the consistency between the mapping table and the shared format.

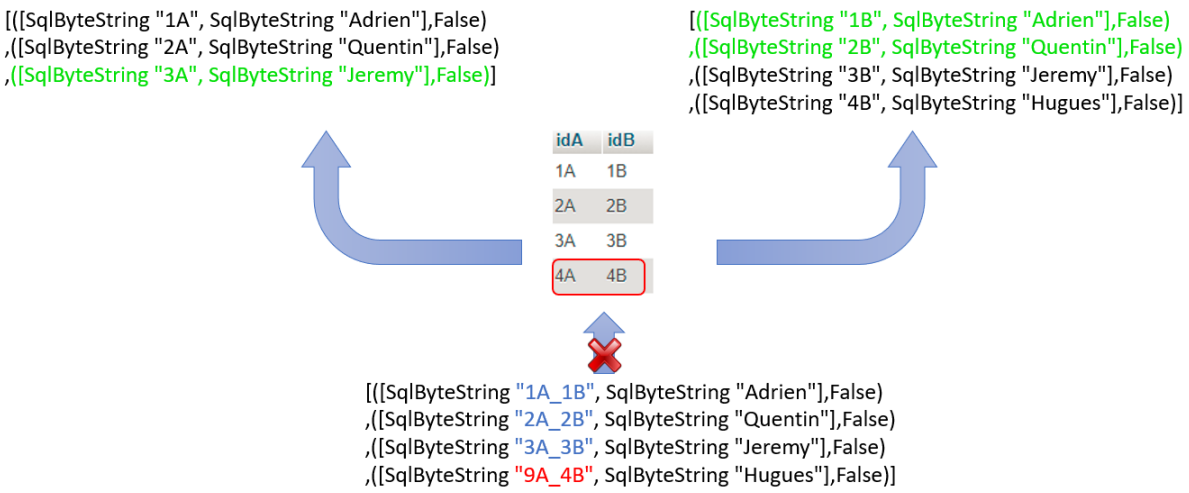

Figure 6.4: Shared Format to Local

6.3 Universal Format Strategy

After discussion and work on the shared format, it seems that it was not really what we wanted to create. We wanted to communicate database informations between peers. The shared format is useful to share data only between two peers. But our first idea was to communicate informations between multiple peers. Then, we decided to think about a new solution with our experience. Our answer is the universal format. We decide to find a new way of sharing. In this section, we will discuss about the universal format. We will explain how we can share data, how it works, ...

6.3.1 Universal Mapping

Just like the shared format, we decided to base our idea on the Hyperion project and its mapping table. For the universal format, we decide to map all the changes into all peers. That means, each peer will keep a mapping table on his own data. As the figure 6.5 shows, the mapping table, on the left, associates the local data into the shared data. Like the shared format, we decide to replace the local primary keys, column names and table name.

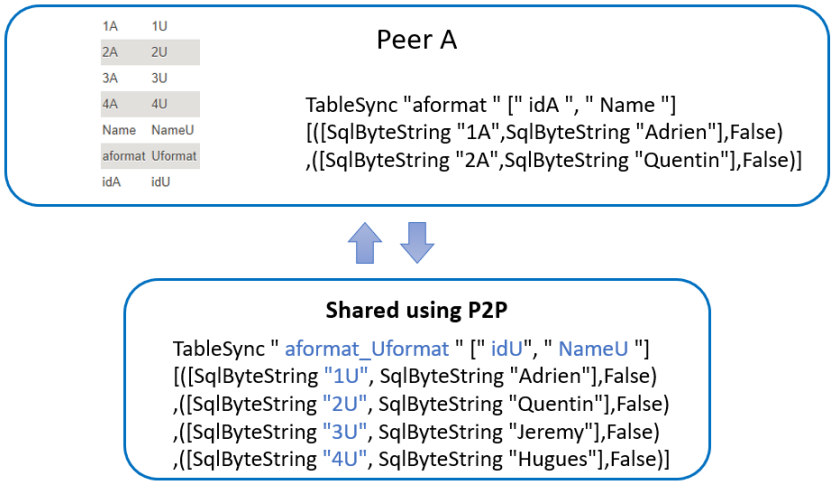

Figure 6.5: Universal Mapping

With this new mapping, it is easier to share data with multiple peers in a same time. As each peer has his own mapping table, a user can add and retrieve data as he wants. Then, different databases can share data easily and keep a local format. For example, let's take three companies. These companies want to sell seats for an event. Normally, seats are shared between companies and each company sell its seats apart. With this system, all companies could sell all seats. The universal format would be the numbers of seat and local formats could be as the peers wants. Let's imagine that in some companies the column for seats is integer, and the numbers of seats are String. The translation between universal and local can resolve this.

As you can see in the figure 6.5, the table name, column names and the primary keys are replaced. This replacement is made with the mapping table. But for the table name, we decide to keep the idea of the shared format. That means merging the data of the universal and local data. Our idea was to know the origin of the first sender of the table. This concept is not completely implemented yet. But, as you can see, the name is *aformat_Uformat* that means the peers who created this universal table contains a *aformat* table.

6.3.2 Local Format and Universal Format

To implement that universal format strategy, we use BiGUL. Bidirectional transformations are the best solution in this situation, because we want to keep the consistency. Here, we want to be able to pass fluidly between the local format and the universal format. Like the shared format, the objective is to translate a list of *TableSync* into another list of *TableSync*. *translateTableSync* is the main bidirectional transformation. The source is the universal format and the view is the local format. Then, the GET function gives us a local format with an universal format and the PUT gives us a universal format.

These bidirectional transformations are very complex. We created a first version with *Case*, as you can see in appendix .2. But the *align* function is the best solution here. With the *align*, it is possible to directly check if everything is defined in the mapping table. Also, some elements in the source are not reflected in the view. Then, a *TableSync* with an undefined name in a local mapping table, will not be retrieved, but all other elements stay consistent. You can see the *align* solution in the listing 6.2.
The first BiGUL function (*translateTableSync*) is a transformation to prepare *translateTableSync2*. With *translateTableSync*, we can just give the mapping table to execute the function. With this mapping table, we will add a boolean in each element of the source to make a couple. This boolean defines if there is a match or not in the mapping table with the *TableSync* name. Then, when we use the GET function, all *TableSync* names with no match in the mapping table will not be reflect in the local view. For the PUT direction, we want to have everything in the source.

```
1  --Some element in the source are not reflect in the view
2  --To keep the consistency, we use a Bool to define what is in the
       view or not
3  translateTableSync :: [[SqlValue]] -> BiGUL [TableSync] [TableSync]
4  translateTableSync mapT = emb g p
5  where
6  g s = fromJust(get (translateTableSync2 mapT) newS)
7          where
8          newS = map (\(TableSync n cols cont)
9          -> (not((findNoUniversal mapT (head (reverse(splitOn "_" n))
               ))/="")),(TableSync n cols cont))) s
10 p s v = map (\(_,t) -> t) (fromJust (put (translateTableSync2 mapT)
       (map (\t -> (False,t)) s) v))
11
12 --translate a TableSync list into a TableSync list with an universal
       format
13 translateTableSync2 :: [[SqlValue]] -> BiGUL [(Bool,TableSync)] [
       TableSync]
14 translateTableSync2 mapT = align
15         -- not deleted item
16         (\(b,_) -> not b)
17         --match between the source and the view
18         (\(_,TableSync nS _ _) (TableSync nV _ _) -> (fromJust (get
               (uniName mapT) nS))==nV)
19         --update function
20         $(update [p| (_,(TableSync n cols cont)) |]
21                  [p| (TableSync n cols cont) |]
22                  [d| n=(uniName mapT); cols=(uniCols mapT) ; cont=(
                        uniCont mapT)|])
23         --create a source from a view
24         (\(TableSync n col cont) -> (False,(TableSync (fromSql(
               idSync (toSql n) mapT)) [] [])))
25         --delete an item from the source list
26         (\(b,table) -> Just (True,table))
```

Listing 6.2: Translation Universal format : main function

The *translateTableSync2* function is a classic align, like it is defined in the first chapter. Here, the condition to match between source and view is the local *TableSync* name. To create a new source, we create a new empty *TableSync* with a name. This name is a special name like explained earlier. The name is created with an association between the local and universal name as it is defined in the mapping table. This is the only time we modify the name of a universal *TableSync*. Also, to delete an item from the peer, we just need to change the boolean value. The table will be conserved in the source and only in the source. For the update function, we decided to divide this into three functions (*uniName*, *uniCols* and *uniCont*).

These three functions are quite the same but with different types. Let's take the most interesting one (*uniCont*), as you can see in the Listing 6.3. The idea is the same as above. The only changes are the match condition and the update condition. The match is done with the mapping table. It will be matched if the universal definition in the mapping table is the same as the source. For the update one, we just choose to manage all the rows with one function (*replaceRow*). This function will just replace the primary key with the mapping table.

```
--Some element in the source are not reflect in the view
--To keep the consistency, we use a Bool to define what is in the
    view or not
uniCont :: [[SqlValue]] -> BiGUL ContentSync ContentSync
uniCont mapT = emb g p
where
g s = fromJust(get (uniCont2 mapT)  newS)
        where
        newS = map (\(row,sync) -> (not((findNoUniversal mapT (
            fromSql (head row)))/="")),(row,sync))) s
p s v = map (\(_,cont) -> cont) (fromJust (put (uniCont2 mapT) newS
    v))
        where
        newS = map (\cont -> (False,cont)) s

--align the content of a Table
uniCont2 :: [[SqlValue]] -> BiGUL [(Bool,(Row,Sync))] ContentSync
uniCont2 mapT = align
        -- not delected item
        (\(b,_) -> not b)
        --match between the source and the view
        (\(b,((s:_),_)) ((v:_),_) -> (findUniversal mapT (fromSql v)
            )==(fromSql s))
        --update function
        $(update [p| (_,x) |] [p| x |] [d| x=replaceRow mapT |])
        --create a source from a view
        (\((v:vs),b) -> (False,(((toSql (findUniversal mapT (fromSql
            v))):vs),b)))
        --delete an item from the source list
        (\(b,cont) -> Just (True,cont))
```

Listing 6.3: Translation Universal format for content

6.3.2.1 Add information

Adding information to share is not a problem. Like we have seen in the previous chapter, the user has select his data and also set the privileges in his database. The PUT of *translateTableSync* will add the selected tables to tables that the peer want to share. This source is stored in a text file. This file will then be shared in a peer-to-peer network. Adding information boils down to merge information in the file using bidirectional transformations.

6.3.2.2 Privilege Management

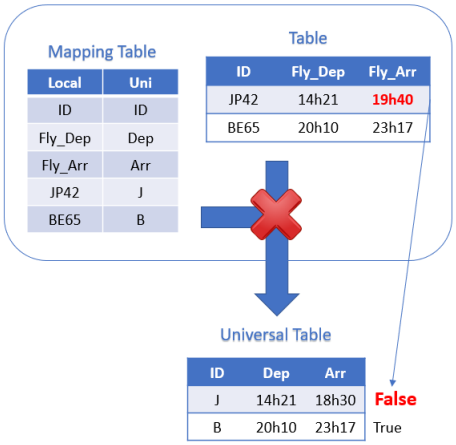

Figure 6.6: Privilege Management

When a user modify information in his database and want to share it, we need to check if the user can modify the data in the network. To do that, we directly use the boolean of the row defined during the selection, as you can see in th figure 6.6. During the PUT direction of the *translateTableSync* function, we will check if there is a modification in the row. If it is the case, then the function will check the value of the boolean. The boolean will allow or not to replace the row in the universal format. If the function refuses to replace the row, we just skip it and notify the user. In the same principle, we decide that if the user is the creator of the table, then he can modify it. For the moment, the creator is defined by the name of the table, as mentioned above.

6.3.2.3 Retrieve informations

When a user wants to retrieve information, the idea stays the same as for the shared format. Only the data defined in the mapping table will be retrieved as pictured in the figure 6.7.

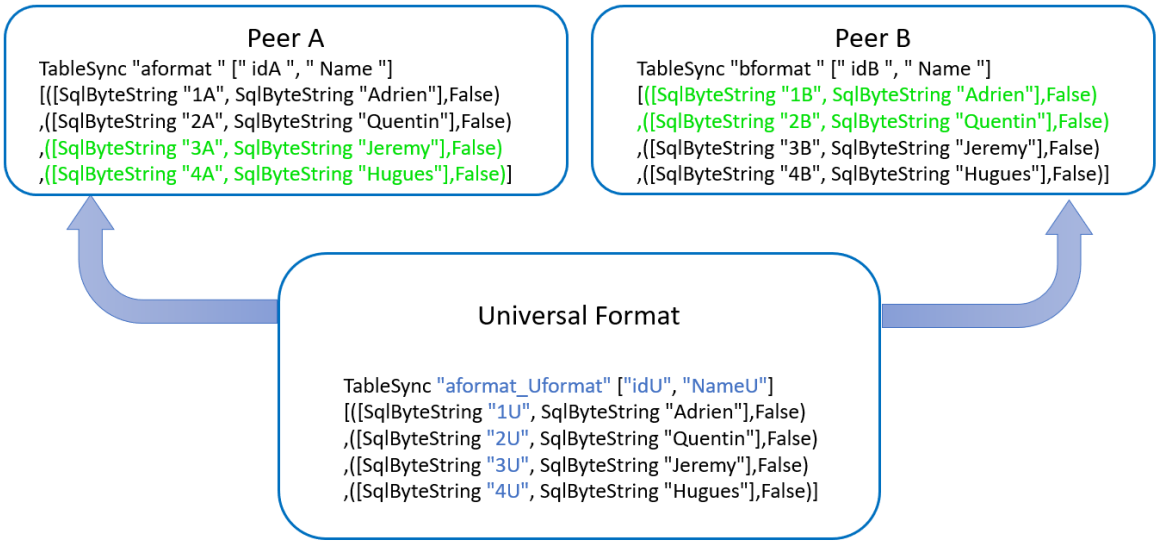

Figure 6.7: Retrieving information with the universal format

6.3.3 Some other functionalities

In this chapter, we explain all we have done about the translation between peers. But there are also useful functions to make that idea real. For example, we mentioned a text file in the chapter. If we have a text file, that means we read and write into it. To retrieve data from this file, we need a Parser to get back all the types. Also, we see that all primary keys we want to share need to be associated in the mapping table. If we have a gigantic Sql table, we hope that we don't need to associate each primary key with.

In conclusion, the universal format strategy is the best strategy to solve our problem. We want to share and retrieve data from a network, and directly integrate it in a database. After the user have selected the data and set privileges, we want to merge and put the data on the network. The universal format strategy is a transformation to allow merging data on a network. Also, when a user wants to retrieve data from a network, this strategy allows to transform the data into a comprehensible format for the peer. The universal format strategy is the link between a universal format shared on the network and the format of each peer.

Chapter 7

Communication

7.1 Overview

In this chapter, the last part of the system is described. In order to share some selected content, the data had first to be retrieved, then consistently selected, to after be also consistently transformed, translated, and it has finally to be pushed on the network. After giving a sketch of the different protocols considered to implement this part, it is concretely explained how the data is pushed and pulled to and from the network, using the Chord protocol. After detailing why it is needed, an alternative to the peer-to-peer systems is explained.

It is important to know that just like the translation part of the system, all the network operations are designed to run in background as no user input should be needed. Also, as this chapter is almost entirely based on the peer-to-peer and the Chord protocols, the reader should clearly have those protocols in mind.

To get more concretely in the system, the main idea behind the network section of the system is the following. The network is considered as a stack of shared data versions retrievable by the members of the sharing process. In this perspective, the network is just a "place" where the data is stored to always be available to anyone querying it.

7.2 Considered technologies

This section presents all the sharing techniques considered. As it is possible to notice, only peer-to-peer technologies are explained. Indeed, in all the previous projects [41, 12, 21, 14], only p2p networks were used, mainly for their characteristics. Indeed, as explained in Chapter 2, the major needed characteristic is the time-shared storage. Also, the load balancing and the advantageous complexity of the lookup are very important aspects. Please note that as p2p networks are used as either centralized, decentralized or fully decentralized directory, and as the time-shared storage property is also used, there must exist a lookup algorithm to be able to locate the data. Obviously, the architecture of the network greatly influences the complexity of this mechanism.

The different technologies considered to ensure the communication between the systems running on different machine are the following : BitTorrent, JXTA, Foreign Data Wrapper (FDW) and finally Chord.

7.2.1 BitTorrent

BitTorrent is mostly known for all the media downloads. Still, its usage was firstly thought to share not only movies, but all kinds of large files. Indeed, the architecture and algorithms used allow to limit the load on the data owner and on the network by sharing parts of the data and reconciling it into a consistent file. To better understand, here is explained the architecture of the system [13].

7.2.1.1 Architecture

First, remembering the architectures explained in Chapter 2, BitTorrent uses a decentralized directory, also known as "hybrid". A central server is indeed used as a "tracker" to make the BitTorrent clients able to exchange data between each other. As it is then understandable, the clients ("leechers") are on a overlay network and are directly sharing data between clients. The tracker stores the location of each client and the files which they are downloading so that other clients can easily join the sharing process and download the complete file from a whole bunch of different clients.

In order to know where the central server is located, a metainfo file called "torrent" stores the URL from which the server can be contacted. Also, this file contains a list of hashes and the name and size of the file to download. As the file to be downloaded is supposed to be huge, it needs to be cut into chunks of data, represented by the hashes. In that case, the clients only have to retrieve the hashes separately and reconcile them using the ordered list in the "torrent" file.

Obviously, to download the file, it firstly need to be completely available on the network. The original downloader of the file, the "seeder" needs to be present on the network at some point in order to distribute the chunks to other clients ("leechers"). The major distinction between a "seeder" and a "leecher" is about the completeness of the file to download.

7.2.1.2 Evaluation

Knowing how BitTorrent works in general, it is now possible to explain why BitTorrent has not been chosen to implement the communication layer of the project.

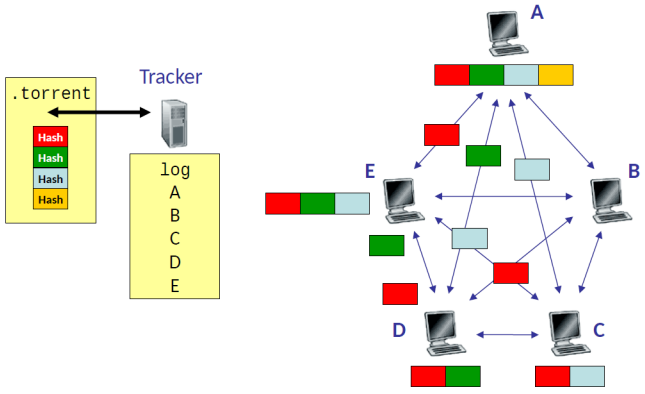

Figure 7.1: An example of the BitTorrent architecture when downloading a file, as in [40]. In this example, A is both seeder and peer while B,C,D,E are peers, or leechers.

First of all, with a peer-to-peer perspective, the "tracker" being a single point of failure or SPOF (if this point fails, the whole system fails too), the method does not really seem as secure as a pure query flooding architecture. On the other hand, the decentralized directory brings a big advantage concerning the lookup time as every location is stored inside the database of the central server.

Second, as in our system, the data selected by the user may contain a huge amount of rows and columns, the file to share may be large too. In this case, the "torrent" file listing the hashes by order is even more critical. Indeed, this system prevents order faults when reconciling the data. If a chunk of data is misplaced, it may be possible that the data will be inconsistent.

On the positive side, an implementation of BitTorrent can easily be found on Cabal [2] and Hackage [3] [20, 43], making the implementation quicker.

Last but not least, BitTorrent requires a central server and a protocol dealing with the overlay network. The two of them together can be costly. A question is then raised : as a central server and *.torrent* files are needed anyway, why not use a traditional client-server architecture easier to implement? Following this logic, BitTorrent has been rejected as component of our system.

7.2.2 JXTA

This technology was not deeply investigated because of its living possibility. Indeed, this communication system has been researched under the supervision of the Sun Microsystems [11]. Yet, this open-source project has been abandoned by the head of research in 2013 due to a lack of dynamism from the development and user community. In that perspective, even though some say that this project is still alive, it has not been considered, as its support is continuously decreasing. Please note that the reference [11] has been partially read because of the decision above.

Despite this fact, a very brief sketch of this technology is given below. JXTA allows to exchange XML messages through a peer-to-peer network notably allowing peers behind firewall or a NAT to participate in the exchange. Each peer is uniquely identified by a 160-bit SHA-1 unique string. XML files are also used by JXTA as "advertisement" and contain information directly about the whole network, like peer groups, pipes,...

It is moreover important to know that this system distinguishes two different kind of peers : the edge-peers and the super-peers which are also composed of two different kind.
The edge peers, as the name indicates, are the one whose bandwidth and connection are low or very limited, often behind a firewall or some other protection.
The Relay peers, one of the super-peers category, are the ones responsible for linking the peers behind a firewall or NAT to the JXTA network.
The Rendezvous peers, the other super-peers category, are responsible for the message exchange, using the "rendezvous" protocol. Indeed, those peers ensure that the messages are delivered using a version of DHT storing the unique identifier of the known peers. More than that, the routing protocol is efficient, notably in the case when a edge-peer publishes an advertisement. In this case, the edge-peer sends the index of the advertisement to the Rendezvous peer it is connected to. This last peer then propagates the index on the network using a DHT function.

7.2.3 Chord

In a certain way, as BitTorrent uses a protocol similar to Chord to deal with its overlay network, it is possible to say that Chord is related to BitTorrent, making a choice between those two technologies quite hard. Still, as it deals only with a fully unstructured network, Chord does not have a centralized server being the SPOF. All peers provide then the same services and have the same responsibilities [40]. Each and every one of them is able to share and receive data, and to update it if authorized.

7.2.3.1 Basic principles

The Chord protocol is used to enhance the efficiency and time of a lookup on the network, thanks to the consistent hashing method [42, 41, 17]. More than lookup, this protocol is also useful to equally share the load of the nodes, still using consistent hashing.

Another concern is the availability of the nodes. Indeed, if a scientist in the USA wants to share data with another one in Japan, it is possible that they will never be available at the same time. Peer-to-peer networks and more concretely the Chord protocol successfully answer this concern. The peers are able to share and receive data while joining and leaving the network at will.

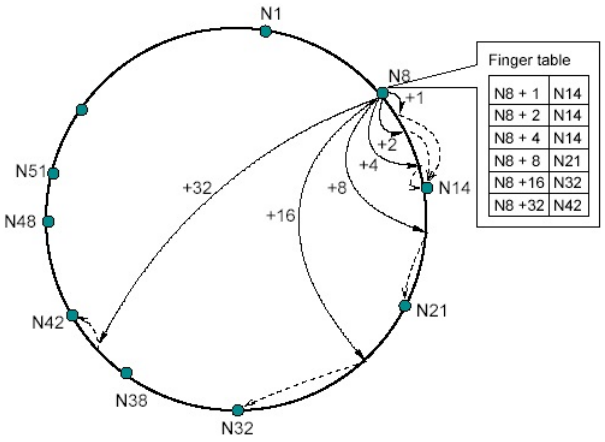

Figure 7.2: An example of fully decentralized network using the Chord protocol. The finger table of node 8 is displayed.

More concretely, the major part of Chord is the consistent hashing. It is indeed useful to give the nodes and keys representing the data to be shared an identifier that is notably able to help balance the load. Then, the hashed keys are assigned to the direct successor node of the key identifier. Referring to Figure 7.2 for example, if nodes "1" and "8" are on the network and key "6" is also, then that key is stored in the distributed hash table (DHT) [38] of node "8". Note that the DHT is a structure stocking both keys and the raw data related. In that case, given a key, each node is capable of finding the node responsible for it by recursively addressing its successors.
Another method consists in maintaining a "finger table" that allows to jump directly to other nodes even if they are not direct successors, reducing then the lookup time. An example of finger table can be found in Figure 7.2. For a node n, a finger table having m entries always follows this formula :

$$successor(n + 2^{i-1}) = s \tag{7.1}$$

Note that s is the identity of the first node and $1 <= i <= m$. So, in Figure 7.2, the left column of the array is the calculation of $n + 2^{i-1}$ while the right column displays s.
It is important to know that besides DHT and finger table, a peer on the Chord circle also maintains a predecessor pointer storing both the identifier and the IP address of the direct predecessor node. This allows to ease the join and leave mechanisms. Also note that as mentioned in [42], using predecessors, it is possible to go counterclockwise on the circle.

7.2.3.2 Join and leave mechanisms

The peers are able to join and leave the network at will.
In order to join the network, a node n needs first to contact another node n' known to be on the network. Then, n' helps n to initialize its predecessor and finger table. When that is done, as n is not described as online, it needs to propagate the availability modifications on the network. Each peer then checks whether its stored routing information needs to be updated or not. Note that even when a node joins, the complexity of the changes propagation remains low as it is, with high probability, $O(log^2 N)$. Only when all the changes are propagated on the network can the node n be transferred the required keys in order to maintain the basic hypothesis that a key is stored in its direct successor node. It is also possible to know that a stabilization protocol is used in order to keep the efficiency and correctness of the successors' list and finger table modification. This protocol will just update the successors' list and let each peer manage the verification of its finger table. This ensures the correctness of the routing information as well as the efficiency of the modifications as all computations concerning one peer are made locally on this very peer.

When a node fails or voluntarily leaves the Chord circle, the stabilization protocol manages to update the successors' list of the peers. Once again, the finger tables will be verified and calculated locally on each peer. Note that if a node fails, the keys contained in that node are not lost. Indeed, Chord uses replication. It means that every key is replicated r times on the network, ensuring the availability of the data.

7.2.3.3 Advantages

As mentioned above, compared to BitTorrent, the major advantage of Chord resides in its fully decentralized architecture. Thus, there is no SPOF, increasing the network robustness. Compared to other decentralized networks, Chord also has the advantage of a lower lookup time and complexity.
Another advantage of Chord is that an implemented version named "Sirkel" can be found on Hackage [32, 34].

Knowing all this, it is understandable that Chord has been chosen as the communication protocol for the advantageous properties it shows.

7.3 Sirkel - a Chord implementation

As stated before, we then decided to use the Chord protocol to manage the communication between systems. We then used Sirkel. Yet, its installation was not so easy, as next section shows. After describing its concrete usage in the system and the link it has with other parts of the system, the raised issues are then discussed.

7.3.1 Installation

As mentioned above, an implemented version of Sirkel can be found on Hackage. We then tried to install this official version. Yet, the installation could not be completed using *cabal* because of dependencies. Indeed, Sirkel 0.1 needs the *haskell98* package installed. Yet, it could not be installed on a *haskell-platform base* superior to 4.7. But the *HDBC-mysql* package, which we struggled to install required a 4.8 *base*. As we saw that a newer version of Sirkel was available on *GitHub* [22], we decided to use this Sirkel 0.2 instead of the version 0.1 available on *Hackage*.

We then tried to install this most recent version of Sirkel by downloading the package from *GitHub* [32]. Following the download, the command *cabal install* was executed in the folder. Yet, because of an error in the signature of the *createTransport* function in the *TestClient.hs* file, Sirkel could not be installed.

While searching how to solve that bug, we found out that *cloud-haskell* may be helping thanks to the different packages it contains. It is a library used to manage connections, whether it is peer-to-peer or client-server. After removing the libraries installed during the failed attempt, we then executed the command line *cabal install cloud-haskell* to add this plug-in to the GCC compiler.
When trying to run the installation of Sirkel 0.2 again, we could see that a new error was raised in the *P2P.hs* file in the *distributed-process-p2p-0.1.3.2* library built with *cloud-haskell*. After a consequent research time and not willing to uninstall all packages, we decided to edit the *P2P.hs* file, the *TestClient.hs* file and the *.cabal* file from Sirkel (it is used to store and install dependencies links).

7.3.1.1 Edition of *P2P.hs*

To modify this file, the package had first to be downloaded from *Hackage* to be manually installed. Then, line 92 had to be modified as the *createTransport* function expected another argument being the couple of *HostName* and *ServiceName*. Below is shown the original code in Listing 7.1 followed by the edited one in Listing 7.2.

```
88 -- | Creates tcp local node which used by 'bootstrap'
89 createLocalNode :: HostName -> ServiceName -> RemoteTable -> IO
       LocalNode
90 createLocalNode host port rTable = do
91     transport <- either (error . show) id
92                  <$> createTransport host port defaultTCPParameters
93     newLocalNode transport rTable
```

Listing 7.1: Original version of the code

```
88 -- | Creates tcp local node which used by 'bootstrap'
89 createLocalNode :: HostName -> ServiceName -> RemoteTable -> IO
       LocalNode
90 createLocalNode host port rTable = do
91     transport <- either (error . show) id
92                  <$> createTransport host port (\sn -> (host, sn))
                        defaultTCPParameters
93     newLocalNode transport rTable
```

Listing 7.2: Edited version of the code

Note that after editing, the downloaded and edited package must be installed using the *cabal install* command in this package directory.

7.3.1.2 Edition of *TestClient.hs*

As for *P2P.hs*, the problem in this file is the *createTransport* function expecting more parameters. This method is used to link the software to the hardware using sockets and, in this case, default TCP parameters.
Again, the missing argument is a couple of *HostName* and *ServiceName*, types of the *Network.Socket* package. Yet, this package is, by default, not imported and the *Network* package is neither in the dependencies of Sirkel. That is why, more than the import in *TestClient.hs*, the *.cabal* file in the Sirkel folder has to be updated to contain the right dependencies when trying to install the *TestClient* file. Below is shown the updated version of *TestClient.hs* in Listing 7.3, where lines from 8 to have been inserted and line 16 has been modified. The updated version of the *.cabal* file is also shown in Listing 7.4 where line 51 has been inserted.

```
 8 import Network.Socket
 9 serviceName::ServiceName
10 serviceName = "8099"
11 hostName::HostName
12 hostName = "127.0.0.1"
13
14 main = do
15         args <- getOpts -- cmdArgs options
16         Right transport <- createTransport hostName serviceName (\
               serviceName -> (hostName, serviceName))
               defaultTCPParameters
17         let rtable = Chord.__remoteTable . Chord.__remoteTableDecl $
               initRemoteTable
18         self <- newLocalNode transport rtable -- ^ this is ourselves
```

Listing 7.3: Edited version of the TestClient.hs code

```
36 Executable TestClient
37 Main-is:            TestClient.hs
38 Build-Depends:      base >= 4.6.0.0 && < 5,
39                     binary >= 0.6.4.0 && < 0.8,
40                     bytestring,
41                     cmdargs,
42                     containers,
43                     distributed-process,
44                     distributed-process-p2p,
45                     hashtables,
46                     network-transport-tcp,
47                     random,
48                     SHA,
49                     sirkel,
50                     transformers,
51                     network
52 hs-source-dirs:     examples
```

Listing 7.4: Edited version of the code

After all those editions and when both *cloud-haskell* and *distributed-process-p2p-0.1.3.2* packages have been installed, Sirkel can finally be installed too using the *cabal install* command inside the Sirkel folder.

7.3.1.3 Only *TestClient.hs*?

When reviewing the system while writing this thesis, we had to reinstall Sirkel. Rather than using *cloud-haskell*, we tried to only edit *TestClient.hs* and the *.cabal* file. It worked. We then concluded that *cloud-haskell* was not necessary at all and that precipitation and a lack of hindsight made us lose a precious time.

7.3.2 Concrete usage

This section explains how the communication is concretely used in the system via the Sirkel package. It is important to note that the implementation of Sirkel itself is not discussed, but can be found in the Annexes. In this section, only the functions interacting with the system are explained in a generic way, the code itself not being of a critical interest.

It is also important to note that the code is described following the user perspective. Indeed, according to the bidirectional transformations, the explanation is expressed in the *GET* and *PUT* direction, i.e. when the user wishes to respectively push or pull data from the network. Please also notice that the *Transport.hs* file dealing with the communication contains more functions than the ones shown below. They can be found in the Annexes. Still concerning the code, it is mainly taken from the *TestClient.hs* file provided by Sirkel and adapted to the needs of our system.

7.3.2.1 Pushing data on the network

As explained in Chapter 3, the user is firstly asked to give the name of the database he wants to use and whether he wishes to pull or push the data. In this case, let us assume that he wants to push the data. As it is possible to guess, the main function of the system is decomposed in way that different functions are called to either pull or push data.

Knowing this, it is possible to understand that, as time is needed to connect to the network, the connection function *launchChordDefault* in Listing 7.3.2.1 is executed as one

of the first commands in the "push" method. We used a default *ServiceName*, which is the socket number at line 2 and *HostName*, which is the IP address of the current host at line 4. Indeed, as it is the development version, those two should not change so much, so it does not have to be written every time. Yet, *launchChord* exists and works exactly the same as this one, just without those default values.
The first step to connect to the network is to use the *createTransport* function to bind the software to the hardware by setting the socket for an IP address. As this function notably returns *Either*, the right element only is extracted if the function terminated without any error. Then, a *RemoteTable* is created in order for nodes to identify known objects such as types and functions. Then, the current node is initiated at line 11 using both the *transport* and *RemoteTable* and then run.
When the process is running, the first step is to bootstrap the connection to an existing Chord node using the list of peers given as parameter. Then, the hash table is created at line 14. It represents the DHT used by Chord to notably store keys. In order to manage this table, Chord needs to deal with blocks, each of them being a chunk of shared data. The line 15 is used for this purpose. When all those steps are correctly done, the process displays a message telling the all the operations are done.

```
1  defaultServiceName::ServiceName
2  defaultServiceName = "8199"
3  defaultHostName::HostName
4  defaultHostName = "192.168.193.138"
5
6  peers = ["192.168.193.139:8199"]
7
8  launchChordDefault peers = do
9          Right transport <- createTransport defaultHostName
               defaultServiceName (\serviceName -> (defaultHostName,
               serviceName)) defaultTCPParameters
10         let rtable = Chord.__remoteTable . Chord.__remoteTableDecl $
               initRemoteTable
11         self <- newLocalNode transport rtable -- ^ this is ourselves
12         runProcess self $ do
13                 bootstrap initState self (map makeNodeId (peers :: [
                       String]))
14                 ht <- liftIO $ HT.new
15                 spawnLocal $ initBlockStore ht
16                 say "Network operations done"
17
18 sendData :: String -> Process ()
19 sendData "" = do say "Cannot send empty data set"
20 sendData data2send = do
21         holder <- putObject data2send
22         say "This is the key of the shared data : "
23         say . show . (map fst) $ holder
```

Now, the system is connected to the network using Chord. During that time, the user needs then to select the data he wants to share. When the data has been consistently translated and approved by the user, it can now be pushed on the network. To do so, the method *sendData* written in Listing 7.3.2.1 at line 18is used. The string taken in parameter is obviously the data to be pushed on the network. If the string is empty, the sending function displays an error message. The data is then put into the DHT using the *putObject* function provided by the Sirkel library. The Chord protocol then manages to push the data at the right place on the network and returns the keys used to fetch this data. The line displays those keys.

7.3.2.2 Pulling data from the network

In the case when the user wishes to pull data from the network, another connection function is used. It combines both connection establishment and data retrieval using a list of keys. It is the role of the *fetchAllDefault* function described at line 1 of Listing 7.3.2.2. As until the line 9, the code is the same as the *launchChordDefault* function described in Listing 7.3.2.1, it is not going to be explained again.

The line 10 is used to read the list of keys from the command line. Note that the list of keys must be entered in the same order that the function *putObject* given by Sirkel has returned it. Otherwise, the data may be meaningless. There is then a certain risk of inconsistency, increasing the need to find an alternative the this keys system.
Then, the state of this node on the Chord ring is snapshot and used to retrieve the whole data matching the list of keys given to the *getObject* function at line 13. Provided by the Sirkel package, this function looks the network up for each key and reconciles the data following the order of the keys in the list. When the data has been merged, the result is returned and used by the *writeFile* function as shown at line 14. As its name indicates, this *IO* function is used to write data in a file which is used to complete the bidirectional "parts" of the system. The function then finishes by telling the user that the system is leaving Chord, meaning that everything can be terminated without any problem.

```
1  fetchAllDefault keylist peers = do
2          Right transport <- createTransport defaultHostName
               defaultServiceName (\serviceName -> (defaultHostName,
               serviceName)) defaultTCPParameters
3          let rtable = Chord.__remoteTable . Chord.__remoteTableDecl $
                initRemoteTable
4          self <- newLocalNode transport rtable
5          runProcess self $ do
6                  bootstrap initState self (map makeNodeId (peers :: [
                       String]))
7                  ht <- liftIO $ HT.new
8                  spawnLocal $ initBlockStore ht
9                  say "launched chord"
10                 let x = read keylist :: [Integer]
11                 liftIO $ threadDelay 5000000
12                 st <- getState
13                 resp <- getObject x (r st) :: Process (Maybe String)
14                 liftBase $ writeFile directory (maybeToString resp)
15                 say "leaving chord"
16                 liftIO $ threadDelay 200000
17                 return ()
```

7.3.3 Raised issues

Even though Chord seems to answer the communication problem in a nice way, it still presents a major issue which has been discovered too late. Indeed, Chord uses unique keys to represent the data on the network. So, in order to retrieve the shared data, the system needs to be provided the keys representing the data on the network. In [42], it is stated that following the 7-layers ISO standard [16], the key exchange process must be done on the application layer because Chord does not deal with it otherwise. As this issue came late in the development process, we have not had the time to find a way with which the keys are consistently shared using Chord. We then just assumed that the key sharing process is external to the system and is thus not dealt with.

Yet, this "dirty" solution does not satisfy the initial concerns because the whole database sharing process is not completely automated, managed and is neither really easy because of those keys. We then came to think of another solution, an alternative to Chord. It is important to know that this alternative has just been thought about, because due to a lack of time, the system still uses Chord and supposes that the keys are shared using another process.

Also, as a key is a hashed chunk of data, if this data changes, the list of keys also changes. In this case, the new keys must always be sent to all the members of the sharing process for them to retrieve the good version of the data. This would be costly.

7.4 An alternative to Chord

7.4.1 Peer-to-peer?

The issues stated above are clear when using Chord. But if BitTorrent was used, it would also be the case. Indeed, the torrent file would also have to be shared by some process which must be different from pushing or pulling the data on the network. It would also have to be managed by the application layer of some system, for example, sent by mail to the members of the sharing process. The edition problem of the data is also the same. A new torrent file would have to be generated and propagated on the network. Those problems are, as far as we know, recurring issues of p2p networks if they are not directly designed for the concrete system, as in PeerCSDB [41] for example.

As during the implementation of the system, before thinking of the communication layer, we successfully used *Dropbox*, the idea of a central server storing the file containing the data to share came to us. This central server would use the Client-Server architecture to avoid the necessity of external sharing processes and files.

7.4.2 Description of the alternative

To answer the edition and communication problems, we thought of a Client-Server architecture where each user would be identified by a unique string, its email address for example. The user sharing the data in the first place would have to notify the system with who this sharing process must be done. He would then have to input the email addresses of all members of the sharing process. This list of addresses would then be used to name the file created and containing the data to be shared in universal format. As the list may contain lots of elements, it would be concatenated and hashed using a reversible hashing algorithm in order to produce a unique file name.

Every known user would have a private folder on the server. When a user pushes a new file on the server, the name of the file would be "un-hashed" in order to clearly identify the users involved in the sharing process. To avoid duplicates, only a shortcut would be created in the user's folder, the real file being stored in another directory. The shortcut would be used to locate the file to be retrieved on the server. If the user is a member of different sharing processes, different shortcuts would be created, each one matching a single process.

It is important to know that first, the name of a file should never change during the whole lifetime of a sharing process, even if the data inside the file changes. Second, when a file is pushed on the network, the server would have to check whether this file already exists or not. Note that if the file already exists, no shortcut should be created, the old file is just replaced by the new one. This allows any user willing to stop being part of a sharing process to delete the shortcut in his folder. The server would then not fetch that file, revoking the user's membership in the sharing process. Note that the file would not be renamed, even if its "un-hashed" name still contains the revoked user.

As the server would be the bottleneck and the single point of failure, several security measure would be taken to ensure the availability of the data. We could imagine to replicate the server, in order to "double" the computational power and increase the availability of the data. A hot swap could also be considered.

It is yet critical to remember that this alternative has only been thought about and never deeply researched neither implemented. The system still uses Chord as the communication protocol.

Conclusion

To summarize, the created system, when the data is pushed on the network, firstly retrieves the whole table to be shared and selected according to user input. Then, the bidirectional transformations coded using *align* selects the exact data to be shared and sets the edition privileges, still according to user input. The selected data is then converted from the local format to a universal one understood by all the members of the sharing process, thanks to bidirectional transformations using *align* and the mapping table. During this process, the modifications authorizations are checked following the edition privileges. Finally, the universal data is pushed on the peer-to-peer network, using the Chord protocol.
When a user wants to retrieve data from this network, he firstly have to give to the system the different keys, which have been generated when the data has been uploaded. The system then retrieves, thanks to the Chord protocol, the data and directly translated into the user's local format thanks to the bidirectional transformation using *align* and the mapping table. The translated data is then merged to the existing data owned by the user. If a row of the retrieved data is not in the local database yet, it is inserted while every existing row is updated if necessary. This merging and creation process is obviously completed by the *align* function in charge of the data selection. Finally, the merged data can be sent to the database following a comparison technique generating requests.

In this paper, it then is understandable that the system solves, using a bidirectional approach, some problems of selective data sharing systems. Indeed, it has been implemented in a way to be more easily handled thanks to the user interaction. Also, thanks to mapping tables used in bidirectional functions, the system is able to limit the knowledge needed by the user concerning all the data structures, which is certainly an advantage. Furthermore, the user has also the power to authorize or not others to edit its shared data, as BiGUL functions can easily deal with such properties. Last but not least, the use of bidirectional transformations enhances the insurance of data consistency and integrity thanks to the proved "well-behavedness" of BiGUL. It is then possible to conclude that bidirectional transformations are here very useful and can be used properly and efficiently in the selective data sharing context.

Concerning the future works, we can see four major aspects that would optimize the system. Please note that the system could nevertheless be polished at some points, notably on the network level.
First, by integrating the foreign keys and multiple primary keys management. Indeed, the system currently only deals with a single-column primary key, which is a very basic case.
Also, the mapping table construction and the universal format could also be improved in order to be automatically generated in a meaningful way. The universal format must effectively carry some semantics to let the user and the system know the matching between universal table, columns and primary keys to the respective local ones. It is then possible that the mapping table and universal format construction may not be fully automatize. Some research must still be done on this level.
In addition to this, even though the displayed information is not complex, an interface

other than a terminal could be more appropriate and ease even more the user interactions. We notably think of a proper GUI.
Finally, the network part of the system could be strengthened in order to solve the key problem. If the peer-to-peer networks were still to be used, a protocol dealing with the keys on each peer and and linking the network level to the transformations level must be integrated. An inspiration may notably be found in the "PeerCSDB" system. Otherwise, the proposed alternative to the Chord protocol can be developed, completely stamping the keys and Chord protocol out.

Bibliography

[1] Apexsql data diff - sql server data compare tool apexsql. https://www.apexsql.com/sql_tools_datadiff.aspx.

[2] The haskell cabal. https://www.haskell.org/cabal/.

[3] Introduction | hackage. https://hackage.haskell.org/.

[4] IO inside - HaskellWiki. https://wiki.haskell.org/IO_inside.

[5] Sql data compare compare and synchronize sql server database contents. https://www.red-gate.com/products/sql-development/sql-data-compare/.

[6] Sql database studio. https://sqldatabasestudio.com/.

[7] Sql server data compare and synchronization tool - devart. https://www.devart.com/dbforge/sql/datacompare/.

[8] How can i synchronize views and stored procedures between sql server databases? https://stackoverflow.com/questions/335805/how-can-i-synchronize-views-and-stored-procedures-between-sql-server-databases, 2008.

[9] How to synchronize databases in different servers in sql server 2008? https://stackoverflow.com/questions/3425651/how-to-synchronize-databases-in-different-servers-in-sql-server-2008, 2010.

[10] Syncing two databases in sql server. https://dba.stackexchange.com/questions/105965/syncing-two-databases-in-sql-server, 2015.

[11] G. Antoniu, L. Cudennec, M. Jan, and D. Mike. *Performance scalability of the JXTA P2P framework*. Campus de Beaulieu, Sun Microsystems, 35042 Rennes Cedex, France and Santa Clara, CA, USA, 2007.

[12] M. Arenas, V. Kantere, A. Kementsietsidis, I. Kiringa, R. J. Miller, and J. Mylopoulos. *The Hyperion Project: From Data Integration to Data Coordination*. Dept. of Computer Science University of Toronto, School of Inf. Technology and Engineering University of Ottawa, Canada, September 2003.

[13] J. Arne Johnsen, L. Erik Karlsen, and S. Saether Birkeland. *Peer-to-peer networking with BitTorrent*. Department of Telematics, NTNU, Norway, December 2005.

[14] A. Bohannon, B. C. Pierce, and J. A. Vaughan. *Relational Lenses: A Language for Updatable Views*. University of Pennsylvania, Pennsylvania,United States, June 2006.

[15] D. A. Bryan, P. Matthews, E. Shim, D. Willis, and S. Dawkins. Concepts and Terminology for Peer-to-Peer SIP (P2PSIP). https://rfc-editor.org/rfc/rfc7890.txt, 2016.

[16] J. D. DAY and H. Zimmermann. *The OSI reference model*. Codex Corporation,, Mansfield, MA,USA, December 1983.

[17] K. Devendra, R. Pravesh, L. Yang, and C. Shashank. *Implementation of chord P2P protocol using bidirectional finger tables.* Department of Computer Science and Engineering, Michigan State University, United States, December 2015.

[18] A. Duchene and H. Marchal. Lightweight data sharing system based on bidirectional transformations. https://github.com/AdrienDuchene/Bx_data_shared.git, 2018.

[19] R. Fagin. *Tuple-Generating Dependencies*, pages 3201–3202. Springer US, Boston, MA, 2009.

[20] farnoy. Github - farnoy/torrent. https://github.com/farnoy/torrent, 2018.

[21] J. N. Foster and G. Karvournarakis. *Provenance and Data Synchronization.* University of Pennsylvania, United States, TBD TBD.

[22] GitHub. The world's leading software development platform · github. https://github.com/, 2018.

[23] S. Hanada. F.34. postgres_fdw. https://postgresql.org/docs/current/static/postgres-fdw.html, 2018.

[24] Z. Hu and H.-S. Ko. *Principles and Practice of Bidirectional Programming in BiGUL.* National Institute of Informatics, Japan, April 2017.

[25] D. J. Boothby. Synchronization of disparate databases. 1995.

[26] N. W. John Goerzen. Hdbc: Haskell database connectivity. https://hackage.haskell.org/package/HDBC, 2005-2011.

[27] Y. Katsis and Y. Papakonstantinou. *View-based Data Integration.* Computer Science and Engineering UC San Diego, University of California-San Diego,La Jolla, CA,USA, 2009.

[28] H.-S. Ko and Z. Hu. *An Axiomatic Basis for Bidirectional Programming.* POPL 2018, Los Angeles, California, United States, January 2018.

[29] H.-S. Ko, T. Zan, and Z. Hu. *BiGUL: A Formally Verified Core Language for Putback-Based Bidirectional Programming.* PEPM 2016, St. Petersburg, Florida, United States, January 2016.

[30] C. Krzysztof, F. J. Nathan, H. Zhenjiang, L. Ralf, S. Andy, and T. James F. *Bidirectional Transformations: A Cross-Discipline Perspective.* Springer, Berlin, Heidelberg, 2009.

[31] D. Leroy Bell, D. Lawrence Cozzolino, M. Lee Lee, and R. Lee Hagen. System and method for database synchronization. (19), 1991.

[32] M. Lysgaard. Sirkel ; a chord dht in haskell. https://github.com/molysgaard/Sirkel/, 2011.

[33] B. Morelli. Guide to data synchronization in microsoft sql server. https://codeburst.io/guide-to-data-synchronization-in-microsoft-sql-server-5c8d65b606c3, 2017.

[34] M. O. L. <morten@lysgaard.no>. sirkel: Sirkel, a chord dht. https://hackage.haskell.org/package/sirkel, 2011.

[35] J. M. Myerson. Cloud computing versus grid computing. https://www.ibm.com/developerworks/library/wa-cloudgrid/index.html, 2009.

[36] K. Nakano, Z. Hu, M. Takeichi, K. Nakano, Z. Hu, and M. Takeichi. Consistent Web site updating based on bidirectional transformation. *Int J Softw Tools Technol Transfer*, 11:453–468, 2009.

[37] B. C. Pierce. Harmony: The art of reconciliation. In *Proceedings of the 1st International Conference on Trustworthy Global Computing*, TGC'05, pages 1–1, Berlin, Heidelberg, 2005. Springer-Verlag.

[38] E. Rescola. Introduction to distributed hash tables. https://www.ietf.org/proceedings/65/slides/plenaryt-2.pdf, AB Plenary, IETF 65.

[39] M. Ripeanu. *Peer-to-Peer Architecture Case Study: Gnutella Network*.* Department of Computer Science, The University of Chicago, 1100 E. 58th Street, Chicago IL 60637, USA, 2001.

[40] L. Schumacher. *Chapitre 2 : Application Layer, pp89 - 100.* University of Namur, Belgium, October 2015.

[41] X. Shen and Z. Li. *Implementing Database Management System in P2P Networks.* School of Computer Science and Engineering, Northwestern Polytechnical University, Xi'an, 710072, China, 2008.

[42] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.* MIT Laboratory for Computer Science, United States, October 2001.

[43] S. Truzjan. bittorrent: Bittorrent protocol implementation. https://hackage.haskell.org/package/bittorrent, 2013.

[44] D. Wagner. SYMMETRIC EDIT LENSES: A NEW FOUNDATION FOR BIDIRECTIONAL LANGUAGES.

[45] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux. Maintaining Invariant Traceability through Bidirectional Transformations.

[46] T. Zan, L. Liu, H.-S. Ko, and Z. Hu. *Brul: A Putback-Based Bidirectional Transformation Library for Updatable Views.* SOKENDAI (The Graduate University for Advanced Studies), Japan and Shanghai Jiao Tong University, China and National Institute of Informatics, Japan, Japan, China, April 2016.

[47] T. Zan, H. Pacheco, and Z. Hu. BiFluX : A Bidirectional Functional Update Language for XML Tao Zan , Hugo Pacheco and Zhenjiang Hu. 2013.

Appendices

.1 Helpers

```haskell
{-# LANGUAGE FlexibleContexts, TemplateHaskell, TypeFamilies,
    OverloadedStrings #-}
module Helpers where

import GHC.Generics
import Generics.BiGUL
import Generics.BiGUL.Interpreter
import Generics.BiGUL.TH
import Generics.BiGUL.Lib
import Generics.BiGUL.Lib.List
import Control.Monad.Except
import Data.Maybe
import Data.List
import Data.Time.LocalTime
import Data.Char
import Types
import Database.HDBC
import Database.HDBC.Types
import Database.HDBC.MySQL
import Data.ByteString.Char8 as ByteS (ByteString,concat,split,pack)
import qualified Data.Map as Map

{----------------------------------------
This file contains all the functions used to help other methods to
    work,
but not specifically related only to those methods.
----------------------------------------}

{----------------------------------------
OPERATIONS ON LISTS
----------------------------------------}

--This method will return True if first parameter is contained in
    the second,
-- False otherwise.
containedIn::(Eq a)=> [a] -> [a] -> Bool
containedIn [] _ = True
containedIn (x:xs) ls = if(elem x ls) then True && containedIn xs ls
        else False

-- returns the list of indexes of all elements in the first
    parameter in the
-- second parameter. I.e. if the element of first list is in the
    second list,
-- the index of the element in the second list is added to the list
    of indexes.
getIndexesList::(Eq a)=> [a] -> [a] -> [Int]
getIndexesList [] _ = []
getIndexesList _ [] = []
getIndexesList (selCol:selCols) cols = if((getIndex selCol cols) ==
    -1)
        then getIndexesList selCols cols
        else (getIndex selCol cols):(getIndexesList selCols cols)


```

```
50 -- Returns the index of an element in the list. If not found, then
       -1
51 getIndex:: (Eq a)=> a -> [a] -> Int
52 getIndex l ls = case elemIndex l ls of
53         Nothing -> -1
54         Just i -> i
55
56
57 --Returns the element at a certain index of a List
58 getElemAtIndex:: ( Eq a) => Int -> [a] -> a
59 getElemAtIndex i ls = case elemAtIndex i ls of
60         Nothing -> error("couldn't find elem")
61         Just el -> el
62
63 -- Returns the Maybe element at a certain index of a list
64 elemAtIndex:: (Eq a) => Int-> [a] -> Maybe a
65 elemAtIndex _ [] = Nothing
66 elemAtIndex i ls =
67         if(i<0) then Nothing
68         else
69                 if(i >= (length ls)) then Nothing
70                 else
71                         Just (getElem i 0 ls)
72         where
73                 getElem :: Int -> Int -> [a] -> a
74                 getElem i count (l:ls) =
75                         if(i==count) then l
76                         else
77                                 getElem i (count+1) ls
78
79 -- This method will return True if the value is at the given index
       in the list of list (in the deepest list), false otherwise
80 valueInDoubleList::(Eq a) => a -> [[a]] -> Int -> Bool
81 valueInDoubleList _ [] _ = False
82 valueInDoubleList val lss index = if(null (filter (\ls -> val == (
       getElemAtIndex index ls)) lss)) then
83         False
84         else True
85
86 -- Returns True if the value at the given index in the first list
       appears at the same index in the list of lists (in the deepest
       list)
87 valueInListAndListList::(Eq a) => [a] -> [[a]] -> Int -> Bool
88 valueInListAndListList [] _ _ = False
89 valueInListAndListList _ [] _ = False
90 valueInListAndListList xs lss index = if(null (filter (\ls -> (
       getElemAtIndex index xs) == (getElemAtIndex index ls)) lss)) then
91         False
92         else True
93
94 -- returns the list of elements of the second list not being in the
       first one
95 elemsNotInBoth :: (Eq a) => [a] -> [a] -> [a]
96 elemsNotInBoth [] bigLs = bigLs
97 elemsNotInBoth childLs [] = []
98 elemsNotInBoth childLs bigLs = removeListCols (getIndexesList
       childLs bigLs) 0 bigLs
99
```

```
100 --Removes an element from a list based on a list of indexes and a
       counter.
101 removeListCols ::[Int] -> Int -> [b] -> [b]
102 removeListCols [] _ cs = cs
103 removeListCols _ _ [] = []
104 removeListCols idxs count (c:cs) = if(elem count idxs)then
       removeListCols idxs (count+1) cs
105         else (c:(removeListCols idxs (count+1) cs))
106
107 --Removes elements from a list of lists based on a list of indexes.
       The indexes
108 -- point at the deepest list.
109 removeListContent :: [Int] -> [[c]] -> [[c]]
110 removeListContent [] cs = cs
111 removeListContent _ [] = []
112 removeListContent idxs (c:cs) = (removeListCols idxs 0 c):(
       removeListContent idxs cs)
113
114 {--------------------------------------
115 OPERATIONS ON SQLVALUE
116 ----------------------------------------}
117 -- Converts SqlValue to CondType
118 setCondType :: SqlValue -> CondType
119 setCondType sqlValue = case sqlValue of
120         SqlByteString _ -> S (fromSql sqlValue)
121         SqlString _ -> S (fromSql sqlValue)
122         SqlInt32 _ -> I (fromSql sqlValue)
123         otherwise -> S ""
124
125 defaultSqlValue :: SqlValue -> SqlValue
126 defaultSqlValue sqlValue = case sqlValue of
127         SqlByteString _ -> defaultSQLByteString
128         SqlInt32 _ -> defaultSQLInt
129         otherwise -> SqlNull
130
131 toCondtype :: String -> CondType
132 toCondtype s = if (False `elem` (map isDigit s))
133         then (S s)
134         else (I (read s::Int))
135
136 toFct :: String -> (CondType -> Bool)
137 toFct (('='):('='):s) = (==(toCondtype s))
138 toFct (('/'):('='):s) = (/=(toCondtype s))
139 toFct (('>'):('='):s) = (>=(toCondtype s))
140 toFct (('<'):('='):s) = (<=(toCondtype s))
141 toFct (('<'):s) = (<(toCondtype s))
142 toFct (('>'):s) = (>(toCondtype s))
143 toFct (('*'):_) = (/=(S "Null"))
144
145 --remove duplicate elements in a list
146 removeDuplicates :: Eq a => [a] -> [a]
147 removeDuplicates = rdHelper []
148         where rdHelper seen [] = seen
149            rdHelper seen (x:xs)
150                 | x `elem` seen = rdHelper seen xs
151                 | otherwise = rdHelper (seen ++ [x]) xs
```

.2 Universal format translation (first idea)

```
uniCont :: [[SqlValue]] -> BiGUL ContentSync ContentSync
uniCont (id:ls) =
Case [
$(normalSV [p| [] |] [p| [] |] [p| [] |])
        ==> $(update [p| [] |] [p| [] |] [d| |])
,$(normalSV [|(\(((s:ss),_):_) -> (s==(head (reverse id))))|]
        [|\(((v:_),_):_) -> (v==(head id)) |]
        [|(\(((s:ss),_):_) -> (s==(head (reverse id))))|])
        ==> $(update [p| x:xs |] [p| x:xs |] [d| x=(replaceRow id);
           xs=(uniCont ls) |])
,$(normalSV [|(\(((s:ss),_):_) -> (s/=(head (reverse id))))|]
        [|\(((v:_),_):_) -> (v/=(head  id)) |]
        [|(\(((s:ss),_):_) -> (s/=(head (reverse id))))|])
        ==> uniCont (lastP id ls)
,$(adaptiveSV [p| _:_ |] [p| [] |])
        ==> (\s _ -> [])
,$(adaptiveSV [p| [] |] [p| _:_ |])
        ==> (\_ (((v:vs),_):_) -> (((toSql(findUniversal (id:ls) (
           fromSql v))):vs),True):[] )
]

uniCols :: [[SqlValue]] -> BiGUL [String] [String]
uniCols (id:ls) = Case [
$(normalSV [p| [] |] [p| [] |] [p| [] |])
        ==> $(update [p| [] |] [p| [] |] [d| |])
,$(normalSV [|(\(s:ss) -> (s==(fromSql(head (reverse id))))) |]
        [|(\(v:vs) -> (v==(fromSql(head id)))) |]
        [|(\(s:ss) -> (s==(fromSql(head (reverse id))))) |])
        ==> $(update [p| (x:xs) |] [p| (x:xs) |] [d| x=(replaceWith
           id); xs=(uniCols ls) |])
,$(normalSV [|(\(s:ss) -> (s/=(fromSql(head (reverse id))))) |]
        [|(\(v:vs) -> (v/=(fromSql(head id)))) |]
        [|(\(s:ss) -> (s/=(fromSql(head (reverse id))))) |])
        ==> uniCols (lastP id ls)
,$(adaptiveSV [p| _:_ |] [p| [] |])
        ==> (\s _ -> [])
,$(adaptiveSV [p| [] |] [p| _:_ |])
        ==> (\_ (v:_) -> [findUniversal v (id:ls)])
]
```

```
--translate a TableSync list into a TableSync list with an universal format
translateTableSync :: [[SqlValue]] -> BiGUL [TableSync] [TableSync]
translateTableSync (id:ls) =
Case [
$(normalSV [p| [] |] [p| [] |] [p| [] |])
        ==> $(update [p| [] |] [p| [] |] [d| |])
,$(normalSV [|(\((TableSync n _ _):_) -> ((head (reverse(splitOn "_" n)))==(fromSql (head(reverse id)))))|]
        [|(\((TableSync n _ _):_) -> (n==(fromSql(head id))))|]
        [|(\((TableSync n _ _):_) -> ((head (reverse(splitOn "_" n)))==(fromSql (head(reverse id)))))|])
        ==> $(update [p| (TableSync n cols cont):xs |]
                [p| (TableSync n cols cont):xs |]
                [d| n=(uniName id); cols=(uniCols ls) ; cont=(uniCont ls) ; xs=(translateTableSync ls)|])
,$(normalSV [|(\((TableSync n _ _):_) -> ((head (reverse(splitOn "_" n)))/=(fromSql (head(reverse id)))))|]
        [|(\((TableSync n _ _):_) -> (n/=(fromSql(head id))))|]
        [|(\((TableSync n _ _):_) -> ((head (reverse(splitOn "_" n)))/=(fromSql (head(reverse id)))))|])
        ==> translateTableSync (lastP id ls)
,$(adaptiveSV [p| [] |] [p| _:_ |])
        ==> (\_ ((TableSync n _ _):_) -> (TableSync (fromSql(idSync (toSql n) (id:ls))) [] []):[])
,$(adaptiveSV [|(\((TableSync n _ _):_) -> ((head (reverse(splitOn "_" n)))==(fromSql (head(reverse id)))))|]
        [|(\((TableSync n _ _):_) -> (n/=(fromSql(head id))))|])
        ==> (\(s:ss) _ -> (lastP s ss))
,$(adaptiveSV [|(\xs -> null(filter (\(TableSync n _ _) -> (head (reverse(splitOn "_" n)))==(fromSql (head(reverse id)))) xs)) |]
        [p| _:_ |])
        ==> (\ss ((TableSync n _ _):_) -> (TableSync (fromSql(idSync (toSql n) (id:ls))) [] []):ss)
,$(adaptiveSV [|(\((TableSync n _ _):_) -> ((head (reverse(splitOn "_" n)))/=(fromSql (head(reverse id)))))|]
        [|(\((TableSync n _ _):_) -> (n==(fromSql(head id))))|])
        ==> (\(ss) _ -> reOrder ss id)
]
```